Apps For All

Coding Accessible Web Applications

by Heydon Pickering

SMASHING MAGAZINE

Imprint

Published 2014 by Smashing Magazine GmbH, Freiburg, Germany.
ISBN: 978-3-94454079-5 (PDF)
Copyeditor and Proofreader: Owen Gregory
Editing and Quality Control: Vitaly Friedman
eBook Production: Cosima Mielke
Tools: Elja Friedman
Syntax Highlighting: Prism by Lea Verou

Apps For All was written by Heydon Pickering and reviewed by Steve Faulkner.

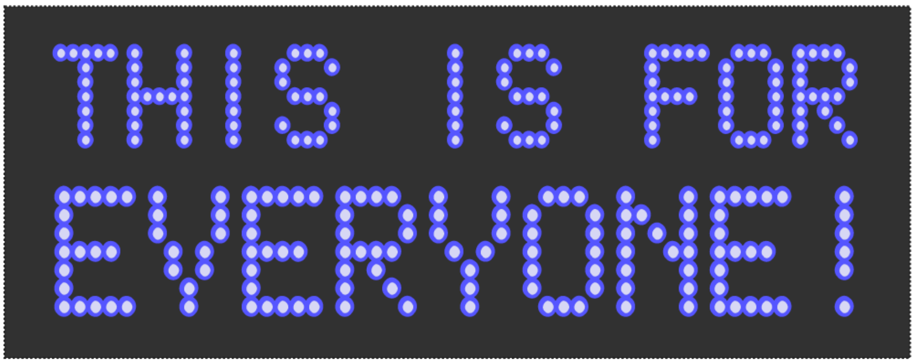
TABLE OF CONTENTS

CHAPTER 1:

This Is For Everyone

At the focal point of a gigantic, purpose-built stadium, in front of a crowd of some 80,000 people and an international television audience of millions, a lone figure, known to many by the enigmatic moniker TimBL, makes adjustments to some electronic equipment.

Given the *nom de plume*, the equipment and the anticipation of a massive audience, one would be forgiven for thinking that this figure is a superstar DJ, preparing to deliver a payload of gut-vibrating rhythms. Instead, the London-born computer scientist, making an appearance at the opening celebrations of the London Olympics, has a simple but enduring message to share: spelled out in LCD lights attached to banks of venue seats, “THIS IS FOR EVERYONE” appears.



It has now been just over twenty years since CERN released the World Wide Web into the public domain[1]. Through the available software and associated technologies of DJ TimBL’s (AKA Tim Berners-Lee’s) invention, we have each been granted the power to contribute to a shared wealth of information. However, making the software to run web servers available to anyone in the world and making the information provided by these servers *consumable* by anyone in the world are two different concerns.

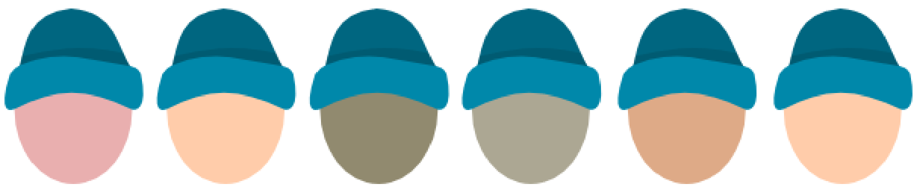
> *The power of the Web is in its universality. Access by everyone regardless of disability is an essential aspect.*
> *— Tim Berners-Lee*

1. http://home.web.cern.ch/about/updates/2013/04/twenty-years-free-open-web

Berners-Lee set up the World Wide Web Consortium (W3C) in October 1994 and left CERN to become its full-time director. Having created a technology destined to be used globally, it was important to make sure new and existing parts of the web would continue to work together correctly so information could thrive. Accordingly, it is the job of the W3C, with the help of participating individuals and organizations around the world, to standardize web technologies such as HTML.

Web Standards

Each year, a number of us celebrate the publication of a book by Jeffrey Zeldman called *Designing With Web Standards*[2], and the blue beanie hat worn by the author on the book's cover has become emblematic of the web standards movement. Standards-based HTML's ability to work with a variety of technologies makes our lives much easier. Moreover, because HTML can be interpreted by a variety of software and devices it is inclusive, or accessible; standard HTML allows content to be seen, read, heard, clicked, typed, and even touched[3], meaning users with different preferences and abilities can consume marked-up content in a way that suits their needs.



It is significant that web standards take the form of a movement rather than a technical criterion. HTML should not simply break whenever it isn't made quite the way it should be. If this were the case, only the highly technically adept would be able to publish content and the medium itself would become inaccessible. Instead, it is our responsibility not to alienate members of our audience and we do this by innovating and sharing HTML authoring practices which prove to be inclusive.

In a TED talk[4] on the evolving technology which helped him to read, Ron McCallum, who lost his sight shortly after birth, does not emphasize the technological side of the web but, instead, appeals to web authors, asking them to use the inherently accessible medium responsibly:

2. http://en.wikipedia.org/wiki/Designing_with_Web_Standards
3. http://en.wikipedia.org/wiki/Refreshable_braille_display
4. http://www.ted.com/talks/ron_mccallum_how_technology_allowed_me_to_read.html

Websites are often very visual, and there are all these sorts of graphs that aren't labeled and buttons that aren't labeled, and that's why the World Wide Web Consortium 3, known as W3C, has developed worldwide standards for the Internet. And we want all Internet users or Internet site owners to make their sites compatible so that we persons without vision can have a level playing field.
—Ron McCallum

THE WEB ACCESSIBILITY INITIATIVE

In the shadow of the web standards movement, few contemporary HTML authors (or web designers) have the audacity to ship or share invalid HTML code. This is a good thing, but there can be a canyon-like divide between a technically valid document — for which automated tests can be instated — and a practically accessible one. Accessible design requires empathy: it is the product of putting yourself in other people's shoes. As Zeldman's seminal book puts it, accessibility is "the soul of web standards."

To address the more oblique nature of web accessibility (compared with basic web standards) as a discipline in its own right, a special branch of the W3C called the Web Accessibility Initiative (WAI)[5] was launched in 1997. Through the creation of guidelines and special technologies, the WAI's primary focus is on making the web a better and easier place for people with disabilities. However, the lessons learned from addressing the specific requirements of those using assistive technologies or consuming information in unusual ways can be applied to enrich the web for everyone. We all win.

Take the TED webpage[6] which hosts the brilliant Ron McCallum talk I quoted earlier. By providing a transcript of the talk's audio in a number of different languages, it not only includes those who are deaf or hard of hearing, but is consumable by people of different nationalities. As an upshot, the content of the talk is better archived and more easily searched (not to mention quoted, as I was able to do a few paragraphs ago). Meeting specific accessibility requirements here has made the presentation of the information therein *better*.

What This Book Will Cover

I think we can agree that web accessibility is really quite a large topic — far too large to fit into a small book. So, what will this small book

5. http://en.wikipedia.org/wiki/Web_Accessibility_Initiative
6. http://www.ted.com/talks/ron_mccallum_how_technology_allowed_me_to_read.html

cover? Though we shall encounter visual design challenges, deal with performance issues, and adopt progressive enhancement — all of which are accessibility concerns — the underlying theme of this book is about making the interactivity of web applications include **keyboard and screen reader users**. Starting by defining simple button controls and moving on to create reusable, accessible widgets, this book is about making interactions possible and meaningful for those who suffer from cognitive and motor impairments, as well as users who experience a range of vision impairments.

TECHNICAL CONTENT

This is a book about modern web application design and, as such, assumes the reader has some familiarity with HTML5 and building interactive applications using JavaScript. JavaScript frameworks like AngularJS[7] will be mentioned, but the patterns we'll accustom ourselves with exist at a level above application business logic, in the DOM[8]. So, there is no affiliation with any particular application design philosophy. Since JavaScript code examples — used to manipulate the state of the DOM when required — will be written using jQuery, a basic familiarity with jQuery syntax is needed.

Throughout, we will pay close attention to the W3C's advice on writing accessible HTML and incorporating inclusive design with CSS. Where the W3C makes recommendations about how JavaScript should harness and interact with these sibling technologies, we shall also take note. Two resources available from the WAI will provide us with most of the specific guidance we'll put into practice. These are WAI-WCAG 2.0[9] (Web Content Accessibility Guidelines) and WAI-ARIA[10] (Accessible Rich Internet Applications). Together, they cover the accessibility of content and the accessibility of *interacting with that content* fairly comprehensively. You do not need to be familiar with either as we embark.

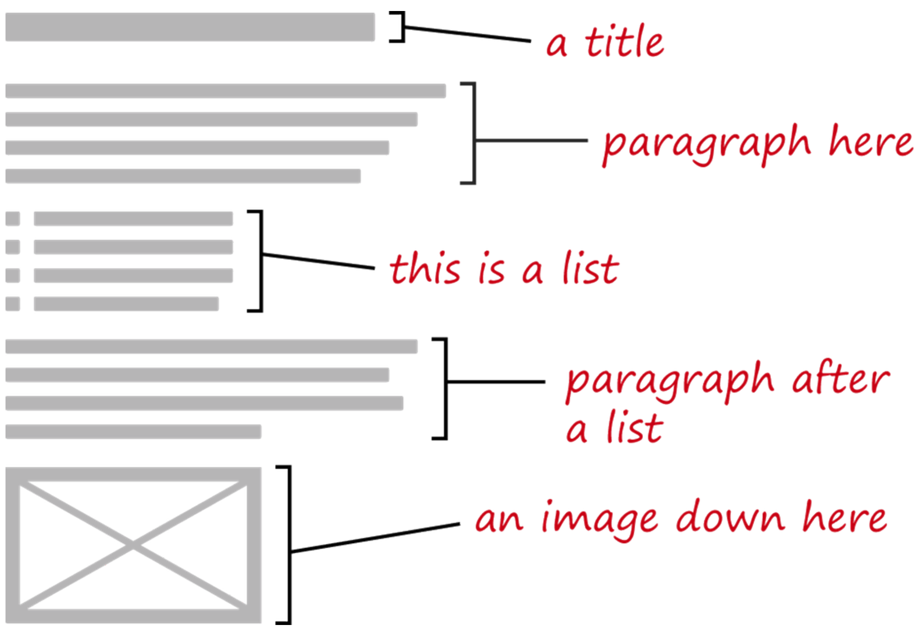
Semantics And Screen Readers

Inescapably, communicating the nature and state of a web application in an inclusive fashion is a question of semantics. The subject of semantic HTML[11] is easily misunderstood and sometimes even maligned. We've all heard the term and many of us have used it, but what do se-

7. http://angularjs.org/
8. http://css-tricks.com/dom/
9. http://www.w3.org/TR/WCAG20/
10. http://www.w3.org/WAI/intro/aria
11. http://en.wikipedia.org/wiki/Semantic_HTML

mantics really *mean* for accessibility? Before moving on, let's take a brief refresher. There's never a good time to talk about semantics, so it might as well be now.

The term *markup* derives from a tradition of marking up manuscripts. This was the practice of providing written notes and instructions in the margins of printed manuscripts to better define or correct their contents. As a type of markup language[12], HTML belongs to a larger category of *metalanguages*. The prefix *meta-* comes from the Greek (μετά-) and its meanings include both *beyond* and *self*. Metalanguages[13] are systems used to help clarify an object language. In other words, metalanguage is language *about* language.



Like their printed antecedents, HTML documents are conduits of human-readable and usable information. That is what they are for. However, words are just words and a page simply filled with a stream of undifferentiated prose would be very difficult to unpick and read. This is where HTML elements come in: by incorporating a helper language we break the seamless prose up into distinct parts, be they titles, paragraphs, lists, or whatever the author intends and HTML allows. These parts can – among other things – then be differentiated visually via CSS.

By providing information about the information, we enrich the basic content of the page and make it more pleasurable to traverse and consume. Take the `<em>` tag, for instance. By encapsulating a phrase within an `<em>` element, we give a better impression of the tone with

12. http://en.wikipedia.org/wiki/Markup_language
13. http://en.wikipedia.org/wiki/Metalanguage

which that phrase should be delivered within a sentence. The tonal quality is encoded by the HTML element, which is represented visually by an italicization of the text within the `<em>` element. Our `<em>` facilitates a typographic substitute for a phonetic feature[14] or, to put it another way, it helps the language come to life.

We're talking about <em>*semantics*</em> now? <strong>**Uh oh**</strong>.

INTEROPERABILITY

Semantic HTML is HTML which makes a positive contribution to the meaning conveyed by the plain language of the page. That is, it makes the semantics known in a standardized way which can be understood by a maximal range of different user agents[15], the bits of software acting on the behalf of users. The ability of different devices and software to use a common language in this way is sometimes called **interoperability**.

Let's look at some HTML which impedes interoperability, and then some which actually helps it.

By defining some CSS classes, you could create what looks like an HTML unordered list, by styling manufactured list items marked up like so:

```
<div class="list-item">First in list</div>
<div class="list-item">Second list item</div>
<div class="list-item">Final list item</div>
```

However, this — combined with the requisite CSS class rules — only tells the browser to render something which looks like a list. By using a standard unordered list (`<ul>`), containing standard list items...

```
<ul>
   <li>First in list</li>
   <li>Second list item</li>
   <li>Final list item</li>
</ul>
```

...we experience a number of benefits. First, this pattern is less verbose. It's also more easily recognized as a known standard by fellow HTML developers. Another advantage is that CSS applied to this list will be ap-

14. https://quote.ucsd.edu/phonoloblog/2006/07/26/phonetics-in-grammar/
15. http://www.w3.org/TR/UAAG20/

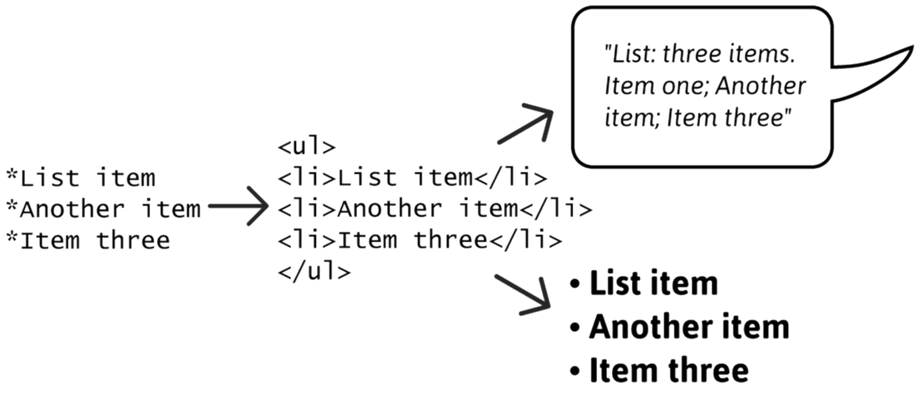
plied equally to any list. This means that standard lists created using WYSIWYG (what you see is what you get) text editors or in Markdown[16] will all enjoy the same styling, keeping things consistent and eliminating the need for non-technical writers to code bespoke HTML directly.

Here's the same HTML list written in Markdown:

```
* First in list
* Second list item
* Final list item
```

The most important advantage for our purposes in this book is that screen readers (or aural user agents as they are sometimes known), will recognize this pattern and communicate it aurally to their user. In effect, the same or similar information is proffered both visually and aurally, making sure users who *look* and users who *listen* have an equivalent experience of the same content.

So this HTML has achieved a pleasing level of interoperability: it can be authored in different ways, using different file formats and conventions; and it can be read and heard, depending on the software involved.



Because CSS class names authored by a web designer are not a standard construct, user agents cannot communicate them in a fully interoperable fashion. That is, if HTML was a form of natural language, then classes would be words which the browser doesn't recognize; a browser can represent those words visually, but it cannot understand them *conceptually*. It is for this reason that the W3C warns HTML writers against relying too heavily on classes to define content:

> *CSS gives so much power to the "class" attribute, that authors could conceivably design their own "document language" based on elements with almost no associated presentation (such as DIV and SPAN in*

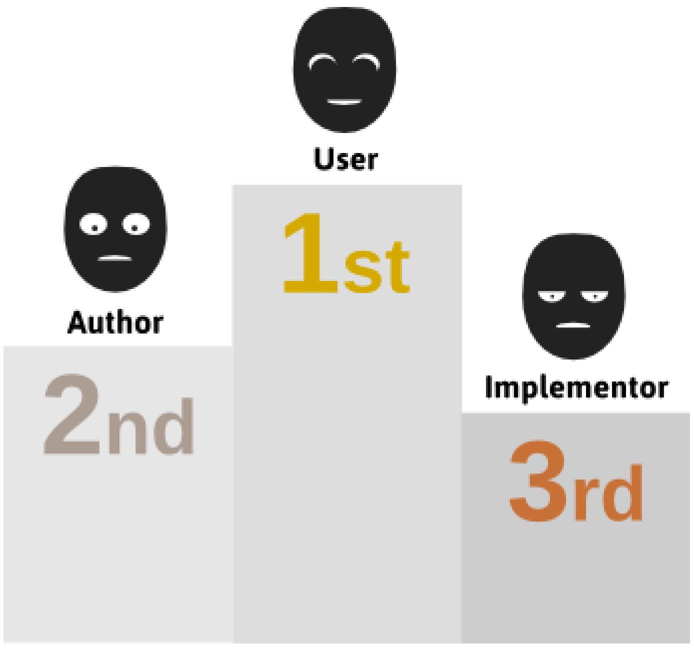
16. http://daringfireball.net/projects/markdown/

> *HTML) and assigning style information through the "class" attribute. Authors should avoid this practice since the structural elements of a document language often have recognized and accepted meanings[...]*
> *– "Selectors", CSS Level 2, W3C*[17]

When you use classes, it is important to use terminology which is understood by your fellow developers. For example, using the class *page-wrapper* to label a `<div>` which wraps the page content is better than using the cryptic *p-w*. However, to paraphrase Harry Roberts[18], this is not a semantic decision, just a sensible one.

According to the W3C's "priority of constituencies", users should be prioritized over your fellow developers. Hence, using sensible class names is less important than using interoperable HTML.

> *In case of conflict, consider users over authors over implementors over specifiers over theoretical purity. In other words costs or difficulties to the user should be given more weight than costs to authors[...]*
> *– "HTML Design Principles", W3C*[19]

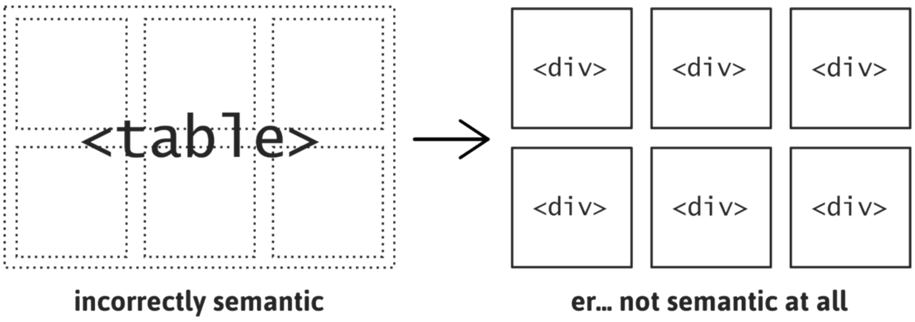


MORE MEANING THAN YOU MIGHT THINK

Semantic HTML is often thought of as using the right element for the job. This is a bit glib. Although an early victory for the web standards movement was encouraging HTML authors to move away from inap-

17. http://www.w3.org/TR/CSS2/selector.html#class-html
18. http://csswizardry.com/2010/08/semantics-and-sensibility/
19. http://www.w3.org/TR/html-design-principles/#priority-of-constituencies

propriately using nested data tables (`<table>`s) for constructing visual layouts, there is much more to semantic HTML than that.



Attributes, as well as elements, have semantic qualities which can be picked up and interpreted by browsers and other user agents. These attributes can affect the visual appearance of elements as well as their behavior. Changing the value of an `<input>`'s `type` attribute from `text` to `checkbox` will affect the visual rendering of the element, the way it is communicated interoperably, and its behavior. In this case, our `<input>` moves from accepting arbitrary text strings to becoming an on–off switch.

As you progress through this book, you will learn about a special set of semantic attributes specified in the WAI's ARIA (Accessible Rich Internet Applications) suite. They are designed to enhance the semantics and, therefore, accessibility of HTML within your web applications. They are an enhancement of and not a replacement for semantic HTML[20].

To begin with, we will spend some time familiarizing ourselves with a generic but strangely unfavored HTML element, without which accessible web applications are simply untenable. ❧

20. http://www.456bereastreet.com/archive/200711/posh_plain_old_semantic_html/

CHAPTER 2:

It's All About Buttons

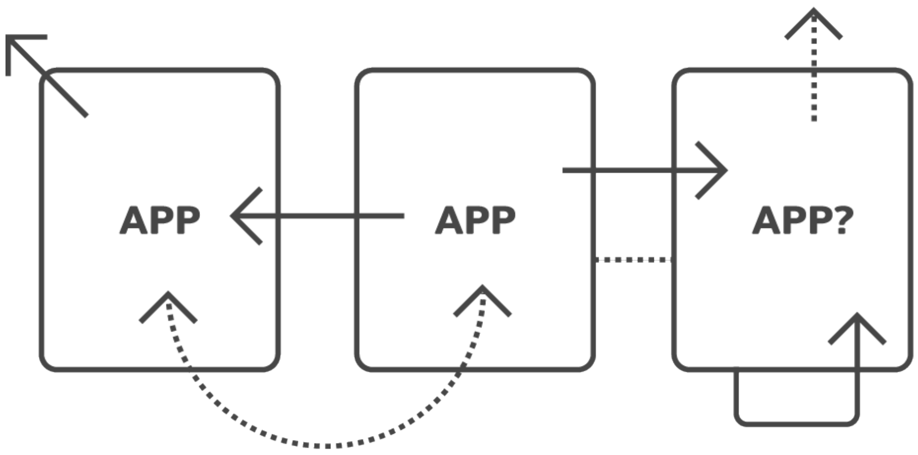
It's scarcely possible to imagine any kind of interface that does not involve buttons of some variety. From fruit machines to ATMs, from futuristic spaceship consoles to iPhone apps, most machines — real or imagined — present their human users with buttons to be pressed at their discretion to make the machine do things.

Even the fabled **BIG RED BUTTON**, with its ominous "DO NOT PRESS" label speaks to us through its familiar form and entices us to touch it — if only to see what falls apart or explodes when we do.

Such is the power of the button.

For many years, the World Wide Web was defined almost entirely by a peculiar type of button called a hyperlink which, when pressed, would take a reader from one location to another in a continuum of interconnected documents. First alphabetically and first in importance, only the `<a>` element could be said to truly define the web: it's the glue which holds all our shared knowledge together.

Increasingly, however, it is web applications rather than simple webpages that have become the subject of our work. More and more we find ourselves constructing and using applications which manipulate rather than simply traverse information. We're even using web applications to construct web applications to build more web applications. In fact, I write this on the day that the ATAG[21] (Authoring Tool Accessibility Guidelines) have reached candidate recommendation. This means provisions for people with disabilities to *author* as well as simply use web applications now have their own, dedicated set of guidelines at the W3C. Progress.



This kind of standardization and the proliferation of JavaScript through easy-to-use libraries like jQuery have made it ever easier for web authors to build so-called one-page JavaScript applications: interactive webpages that are defined not by being connected but by being isolated. If the hyperlink takes you places but a JavaScript application is self-contained, what role does the hyperlink even have?

Truly, there's no such thing as a *pure* web application and, as we shall see, hyperlinks still have an important navigational role within and around application pages. However, the buttons that **make things happen** within our application will be the ones that define it.

Accessible applications start at the press of a `<button>`.

The <button> Element

Some web designers get annoyed with the W3C for providing confusing advice. Out of these designers, I imagine only a few have actually read whole parts of the HTML specification[22] but it's true that the finer points of HTML authorship are occasionally difficult. That said, the specification for `<button>` is hardly a case in point!

21. http://www.w3.org/WAI/intro/atag.php
22. http://www.w3.org/TR/html5/

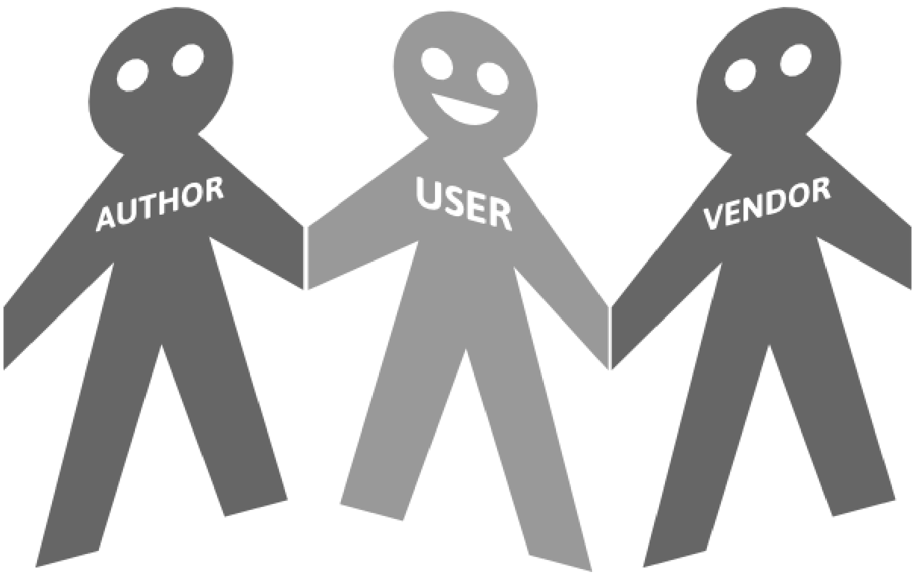
The button element represents a button.
— W3C[23]

In the course of this guide we shall look at some of the many ways `<button>` elements can be enhanced with WAI-ARIA states and properties to express detailed information about their different purposes in an accessible fashion. We'll look at pressed and unpressed buttons, buttons which control other elements, and buttons that show and hide different parts of the application in question. For now, your basic button — the thing that makes stuff happen in your application — should look something like this:

```
<button>Make something change</button>
```

HOW IS THIS ACCESSIBLE?

It's easy to think of using proper semantic HTML as fussy and overparticular; that using the right element for the job is not really that important. Since you can attach JavaScript events to any old element and you can make any old element look like a button with CSS, isn't which element you use a bit academic?



It can seem that way, but no. You see, web standards are all about agreement. It's only through agreement that things can be made to work and behave in ways that are predictable for the greatest number of people. By designating certain behaviors to the `<button>` element, browser vendors can agree on how the element should be rendered and

23. http://www.w3.org/TR/2011/WD-html5-20110525/the-button-element.html#the-button-element

how it should behave. This way, authors like you and I will know which element to code if we want to *elicit* these behaviors. We work with the browser vendors to make our users' lives easier. By convention, they know what they're getting when they encounter a button.

> *If Assistive Technology vendors do not work with developers and browser vendors by making use of the information provided through standardized interfaces for making HTML content accessible, it's the user that gets screwed.*
> *— Paciello Group Blog*[24]

In the case of buttons, two important groups of users benefit from the `<button>` element being used properly as a button control. These are keyboard users and screen reader users. Three main features come free when you use `<button>`:

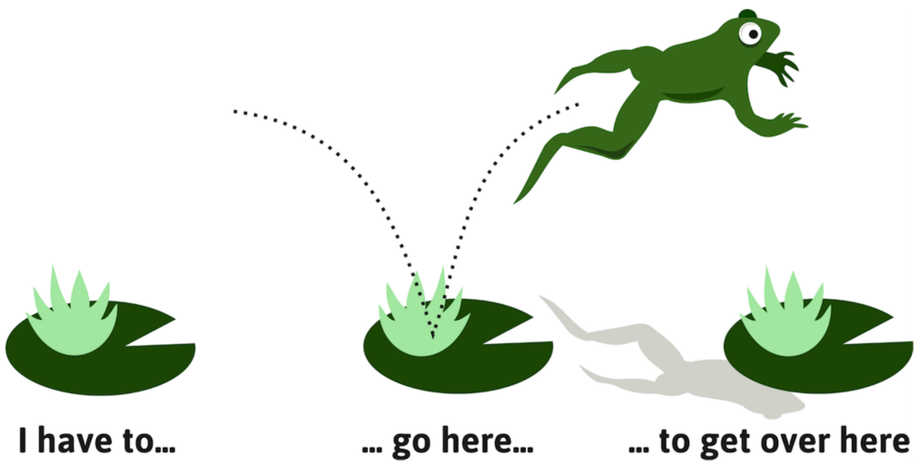
- Buttons can receive focus.
- Buttons can be operated conventionally by the keyboard once focused.
- Buttons are announced as "button" by screen readers.

Keyboard users navigate webpages using the keyboard instead of a mouse. They might do this out of preference, but many do it out of necessity: it's not easy for some people to point the mouse accurately or even see the tiny cursor arrow on the screen to know where it is. Making webpages usable by keyboard is part of WCAG 2.0 (the Web Content Accessibility Guidelines, published by the Web Accessibility Initiative).

> *Make all functionality available from a keyboard.*
> *— WCAG 2.0 Guideline 2.1*

For keyboard users to activate a button, they must first be able to focus it. Focusing an element is like holding it. In a webpage, pressing the *Tab* key will focus the next interactive element (perhaps a button) in the page. The *Enter* key or space bar will then activate it and trigger the event. That event could take any number of forms, but it's how the event can be *accessed* that is our concern here. If the user is not able to hold the element, its action is not accessible.

24. http://blog.paciellogroup.com/2013/11/short-note-aria-dragon-accessibility/



Screen reader users tend to use keyboards as well, but they can't see the button being focused. The screen reader needs to know when to tell them whether the element they've focused is a `<button>` and marking up the button as a `<button>` is by far the easiest way to go about this.

"button: save"
Save

BUTTON TYPES

Given all the lovely common sense stuff we've discussed about `<button>` so far, it's astonishing they aren't used more often as button controls in our applications. How many times have you inspected a web app to find the buttons in it are really `<a>` elements or — even worse, because they can't receive focus by default — `<span>`s and `<div>`s?

The main reason for this is that `<button>`s are associated with HTML forms, making designers fearful that a `<button>` necessitates the presence and submission of a form.

It's true that `<button>`s can be used in combination with forms and other form elements. Button elements even take a `form` attribute to associate them with a form's `id`. In addition, two of the three `type` attribute values for forms — `submit` and `reset` — are explicitly for use within forms. The evidence is stacking up.

However, a further `type` value of `button` is also offered. Buttons with this type don't have to have anything directly to do with forms. In fact, the specification states that, in this regard, `<button type="button">` should:

> *Do nothing.*
> *— W3C*[25]

So, what we have here is an accessible, keyboard-triggerable control unencumbered by form submission or page reloading. It is the most suitable markup for controls within a modern, accessible JavaScript application and we shall be using it liberally. Here's a quick recap about button types:

`submit`	Submits forms
`button`	Good for JavaScript events
`reset`	DANG! WHY DID YOU PUT THAT IN THERE?

test.css

Most browsers will treat a simple `<button>` that exists outside a `<form>` in the way that most authors intend it: as something that *does JavaScript*. However, the implicit `type` for `<button>` is `submit` and some browsers will assume that a button inside a form without a `type` attribute is meant to be a submit button. Not to worry, we can test for this!

Throughout this guide, we shall be using a special style sheet called *test.css* to check for vulnerabilities and accessibility problems. By using CSS selectors we can define and identify bad patterns in our HTML. In this case, we want to check that we've put an explicit `type` attribute on any buttons that go inside our forms. Using pseudo content, we can warn ourselves with an on-screen message if we've messed up.

Add the following declaration block to your *test.css* file and include it as the last style sheet in your page's `<head>`.

```
form button:not([type]):after {
        background: red;
        color: white;
        content: 'Warning: this button doesn’t have a type
```

25. http://www.w3.org/TR/2011/WD-html5-20110525/the-button-element.html#the-button-element

```
        attribute. Is it a submit, reset or just button?';
}
```

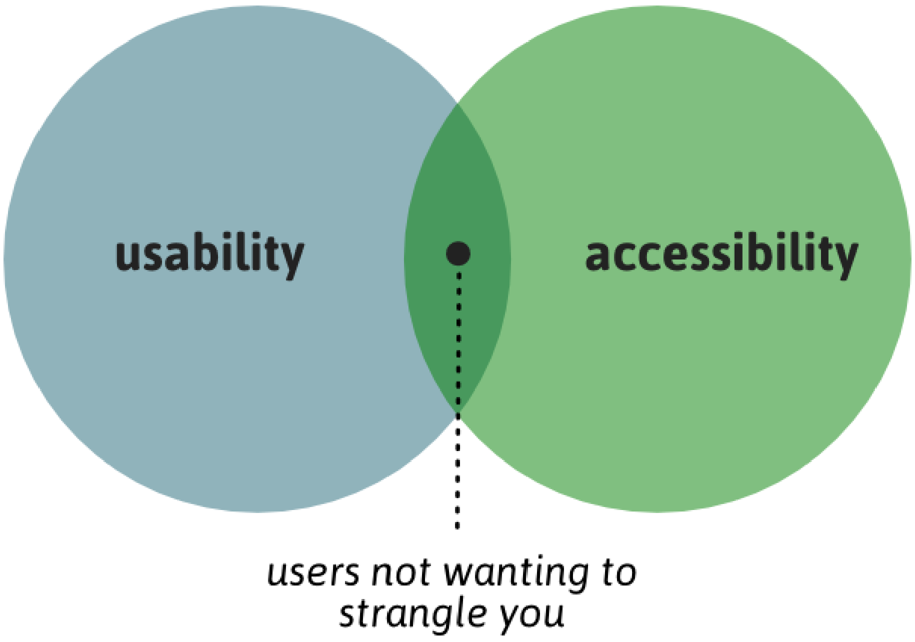
You can change the appearance of the warning however you wish. I've just made it red with white text.

The Style Of Buttons

> *Design is a plan for arranging elements in such a way as best to accomplish a particular purpose.*
> *— Charles Eames* [26]

Now that we have the correct code to make our application buttons, it's time we concentrated on making those buttons appealing. I'm not talking about demonstrating good taste (although extremely ugly buttons would probably be counterproductive) but about ensuring the buttons appear and change in appearance according to users' expectations. They should appeal not just to users' aesthetic prejudices but to their prior understanding or cognition of what buttons are.

Cognitive accessibility[27] is a field which helps everyone, not just people with clinically identified difficulties. It is where accessibility and that other noble goal, usability[28], could be said to converge.



---

26. *Eames Design: The Work of the Office of Charles and Ray Eames* by John Neuhart, Charles Eames, Ray Eames and Marilyn Neuhart, 1989
27. http://webaim.org/articles/cognitive/
28. http://en.wikipedia.org/wiki/Usability

Because HTML is "for everyone", authors should be able to create usable documents and applications without having to be great artists. For this reason, `<button>`s which appear in documents without CSS associated with them look like buttons anyway. It's not that they're unstyled at all; the browser just makes them look OK on the author's behalf. This way, any author can make a usable webpage without having to learn the additional technology of CSS.

Allowing form to follow function in your button design makes it easier for users of all cognitive abilities to use your app or website. This is why browsers will render `<button>`s to look like buttons by default. When restyling buttons, it's best to take cues from these conventions.

BUTTONS YOU WANT TO PRESS

At the time of writing, a trend that has come to be known as "flat design[29]" has reached a peak of popularity and controversy. The flat design mode dispenses with the shadows, gradients, and textures that have become the fabric of our app and website interfaces.

Designers the world over have adopted flat design because it provides a striking and stylish finish to our applications' appearance and helpfully reduces clutter. Either that, or because they've got caught up in flat design as an arbitrary trend. In any case, with flat design you do have to be careful. Interaction design is about things you can use, not just admire, and the perceived tactility of interactive controls can help to communicate their utility.

In the following example I aim to make my button "button-like" by employing a simple `border-radius` and a `box-shadow` to raise it slightly from the page. I know it's not raised literally because the screen itself remains flat, but it should look like it is.

```css
button {
        background-color: DarkSlateBlue;
        border-radius: 0.25em;
        box-shadow: 0 4px 0 #222;
}
```

29. http://www.smashingmagazine.com/2013/09/03/flat-and-thin-are-in/

Size and Contrast

Don't be ashamed of your buttons. Make them big and bold. This makes using controls easier for those who have trouble controlling the mouse input but have not resolved to use a keyboard instead. This is the reason inputs with a `type` of `checkbox` can be wrapped in a `<label>` element which will dutifully act as the expanded target area for clicking that input.

```
<label><input type="checkbox" value="Yes" /> Do you agree?</label>
```

> *The ability to click or press a label to trigger an event on a control provides usability and accessibility benefits by increasing the hit area of a control.*
> *— Emphasis mine, "The label element", W3C*[30]

In the case of buttons, it is common to design them as self-contained boxes filled with a background color. So that one, familiar color is used to signify things the user can press, you may want to make this background color the same as the text color of your other principle controls: links. That isn't a requirement, but you are required to make the contrast between your text and background color sufficiently high.

> *Make it easier for users to see and hear content including separating foreground from background.*
> *— WCAG 2.0 Guideline 1.4*

Save
homepage

A quick way to check that your button contrast is viable would be to enter your foreground and background colors into Lea Verou's simple tool[31]. More tools like this one for color testing (and color blindness testing) are listed in "Welcome To The Community", chapter 7 of this book.

30. http://www.w3.org/html/wg/drafts/html/master/forms.html#the-label-element
31. http://leaverou.github.io/contrast-ratio/

THE STATES OF A PRESSABLE BUTTON

Buttons can be said to be in different states depending on how far you've got with actually pressing them. Whether you are waving your cursor over the button (`:hover` state) or focusing it via the keyboard (`:focus`), you can consider these equivalent states of readiness to press. Because they are equivalent states for different types of user, I suggest you take Roger Johansson's advice[32] and combine the rules in your style sheet:

```
button:hover, button:focus {
	/* make it look like you can press it */
}
```

By default, most browsers will also apply some sort of an outline (`outline: thin dotted`, for instance) to hyperlinks to indicate they are focused. It is important this rule is not removed unless you are sure to put something at least as visible in its place. The dotted outline is effective for links because of the irregular shape of the text, but a dotted outline is not much use for buttons because the line sits too snugly around the shape of the box, making it indistinct.

> *Any keyboard operable user interface has a mode of operation where the keyboard focus indicator is visible.*
> *— WCAG Guideline 2.4.7*

If the button is square, you can improve the visibility of the `:hover` and `:focus` states using a thicker, solid outline. Otherwise, a change in `background-color` can help. To achieve something more ambitious, you could use `outline` after all, and animate the little known `outline-offset` property…

Animated Button Outlines

To help draw the attention of keyboard users to newly focused button controls, wouldn't it be nice to gradually close in on those controls? By using a CSS transition to decrease the `outline-offset` property, we can cast a wider net of visibility before pinpointing the button's exact position.

To set up this technique, we need to set an initially wide `outline-offset` and make it invisible. We don't want big, random boxes all over the page. We also need to define the transition type.

32. https://twitter.com/rogerjohansson/status/382531860686848000

```
button {
        outline: 2px solid transparent;
        outline-offset: 100px;
        transition: 0.5s all ease;
}
```

Then we just need to tighten the net while simultaneously making the outline visible.

```
button:focus {
        outline: 2px solid #000;
        outline-offset: 0;
}
```

For users whose browsers do not support `outline-offset` (Internet Explorer, predictably), a two-pixel outline is still visible on focus and fades into visibility. The poor souls who don't get CSS transitions or `outline-offset` still get that chunky two-pixel outline. In other words, the technique degrades gracefully[33]), meaning a minimal number of users are marginalized.

This shrinking outline technique[34] may not suit your needs or tastes, and your users may not accept it as helpful, but the point is this: accessibility doesn't have to be a chore, something you just have to do when the fun parts of designing an interface are over. Sometimes researching improvements to accessibility can be creatively rewarding too.

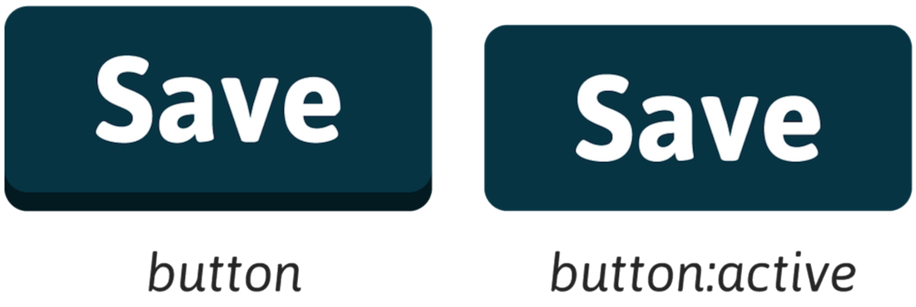
The Active State

I'm going to sound like a creep with a button fetish, but it is rather nice when the button *depresses* suggestively under one's touch, no? Cognitively speaking, it is the expected behavior of a pressed button which

33. http://www.css3.info/graceful-degradation/
34. http://www.heydonworks.com/article/shrinking-button-outlines

started out unpressed and standing proud. This is where the `:active` state comes in. Building on my initial example, I'm going to create the illusion of a real pressed button by reducing the `box-shadow` by the same amount I move the button down.

```
button {
        position: relative;
        top: 3px; /* 3px drop */
        box-shadow: 0 0 0 #222; /* less by 3px (to zero) */
}
```



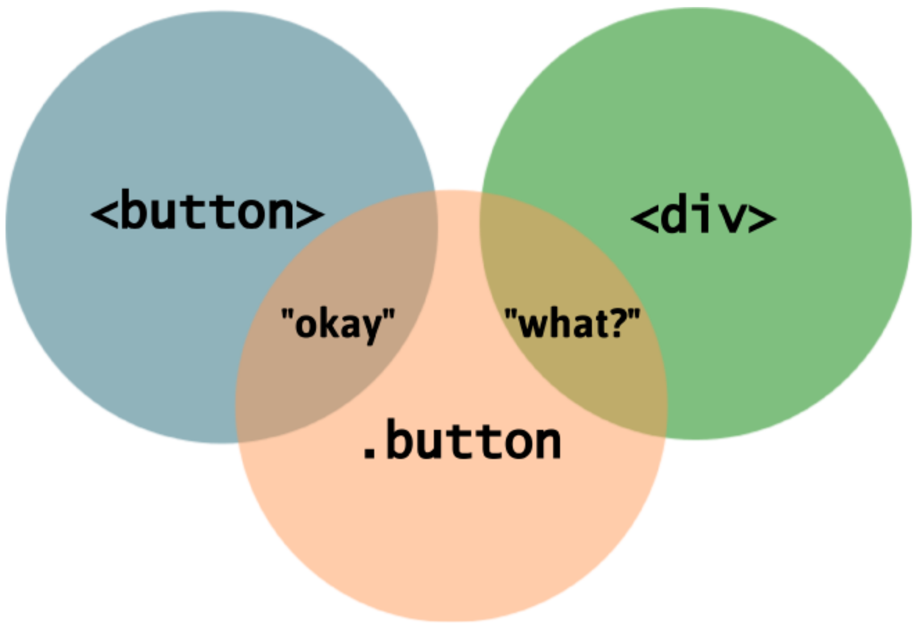
HOW TO STYLE HTML ELEMENTS

Didn't we just cover this? It depends on what you mean by *how*. Yes, we've covered how our buttons should appear, but not how we actually make them this way. As I shall explain, this is an important factor confined not just to the CSS of buttons but all varieties of elements.

CSS frameworks[35] tend to base their style rules on CSS classes. The reason for this is that a CSS framework that contains lots of HTML can't really be considered a pure CSS framework. By using classes like `.button` in their style sheets, they are able to define the look of a button in an abstract way. It's then up to the author to use the framework's `.button` class on elements they see fit.

The problem with this is when accessibility comes in. As we have established, only true `<button>`s should look like buttons because they *represent certain behaviors*. Making `<span>`s, `<a>`s and other elements look like buttons is deceptive. Nonetheless, the cipher of `.button` allows us to make this mistake.

35. http://usablica.github.io/front-end-frameworks/compare.html



CSS classes are neither inaccessible nor accessible, but they are not conducive to accessibility. By confining our button styles to the `<button>` element instead, we ensure that anything that looks like a button can be expected to behave like one.

```
.button {
	/* these styles could go on anything :-/ */
}

button {
	/* styles for correct buttons only */
}
```

Disabling Buttons

As I have established, accessible HTML isn't just about elements but also element attributes, and we shall look at the special ARIA attributes as a particular point of interest. Attributes can define the appearance but also the behavior of the elements to which they are attached. In the case of the `disabled` attribute, we want to take away most of the characteristics which make an enabled button what it is. We are disarming it as we would a gun by removing its ammunition.

As with `.button` there are pitfalls in using a class to make a button appear disabled. A `.disabled` class or `.off` class — whatever you choose — will only make something *appear* disabled. The separate `disabled` attribute still has to do the hard work.

There are two inherent dangers in using classes to disable elements visually.

1. Not all elements can be disabled.
2. Classes don't properly disable the ones that can.

That's pretty useless. The only way you can disable a button is by including the `disabled` attribute on the element and the only way to disable a hyperlink is to remove the `href` attribute. Accordingly, your CSS could look something like this:

```
[disabled], a:not([href]) {
        /* styles for any element that takes a disabled attribute
        or links that do not have the href attribute present */
}
```

test.css

Links without `href`s, like buttons which include `disabled`, do not and should not receive focus. This does not stop confused developers using a `.button` class on `<a>` elements without `href`s, thinking they have created a functional, enabled button. If that last sentence made no sense to you, don't worry: it probably means you're on the right track. You wouldn't believe the number of times I've seen (or made, alas!) this mistake, though.

```
<!-- Not a functional button but a disabled link -->
<a class="button">Press me</a>
```

We're going to add some new rules to our *test.css* file now, to check that we haven't put a `.button` class somewhere inappropriate and to make sure we're disabling things correctly.

```
.button:not(button):after {
        background: red;
        color: white;
        content: 'Warning: You are making an element look like a
        button here. Is it really a button?';
}

button.disabled:not([disabled]):after, a.disabled[href]:after {
        background: red;
        color: white;
        content: 'Warning: It looks like you are styling an
        element to be disabled here. Make sure it is disabled
        properly.';
}
```

Labeling Buttons

We've got so carried away with the design of buttons that we've forgotten some people can't see them at all. How do we communicate to these people (and everyone else as well) what each particular button is for? The simple answer is: with words; the more technical answer is: via the `<button>` element's text node. WebAIM's guide to accessible forms[36] couldn't be clearer about the importance of including text nested within a button:

> *The value attribute for input buttons and the nested text for* `<button>` *elements will be read by screen readers when the button is accessed. These must never be left empty.*

Screen readers will announce the button's label, usually after announcing that the focused control is indeed a button. For example, "Button: Save".

Unless you are deliberately trying to manufacture some sort of guessing game, the announcement merely of "button" is not going to win you many accessibility plaudits. For example, when testing an iPhone app for Southwest Airlines[37], Victor Tsaran encounters a screen announcing just "Button" for each of the controls. "OK. I guess no business for Southwest," he wryly concludes.

However, the text node is not the only way to label a button. It probably should be, but it isn't. Inside the `<button>` you could include an

36. http://webaim.org/techniques/forms/controls#button
37. https://www.youtube.com/watch?v=StIoiIufJzk

`<img>` element instead, perhaps representing a recognizable symbol such as a curved, left-pointing arrow to represent an undo action.

If this is the case, you must make two provisions:

1. Include a tooltip so users who (inevitably!) don't understand your symbol have a textual hint.
2. Include an `alt` attribute to provide a textual interpretation of the button to screen reader users.

```
<button role="button"><img src="undo_icon.png" alt="undo"
/></button>
<p class="tooltip hidden">Undo</p>
```

Note: Using the standard `title` attribute would be a more efficient substitute for a JavaScript-manipulated tooltip, but `title`s are only revealed on hover, not focus, making them inaccessible to keyboard users. Of course, the chances are that whoever made the tooltip library you're using didn't think of supporting the focus event either but, hey, you can fix that...

```
button:hover + .tooltip, button:focus + .tooltip {
        display: block;
}
```

We'll be exploring more accessible, WAI-ARIA-based tooltips in "Peekaboo", chapter 5.

It's important that the `alt` text is what a standard text label *would* read. So, "Undo" is good but "Picture of an undo arrow" is rubbish. It's an alternative way of expressing the function, not a description of something someone can't see.

LABELING WITH ARIA

Provide text alternatives for any non-text content so that it can be changed into other forms people need, such as large print, braille, speech, symbols or simpler language.
— WCAG 2.0 Guideline 1.1

It's time for our first encounter with the Web Accessibility Initiative's ARIA suite. ARIA (Accessible Rich Internet Applications) provides two methods for accessible text in the form of `aria-label` and `aria-labelledby`. The first is an attribute which contains the label text in its value and the second refers to another element which contains the text. ARIA offers these properties to assist screen reader accessibility by providing additional or missing text content to be read.

We shall look more closely at ARIA properties and their contribution to web application accessibility in more detail in the next chapter.

aria-label

In the following example, we'll use a special Unicode character to render an icon from an icon font. The Unicode code point exists within the Private Use Area[38], so has no designated meaning and cannot be announced. The `aria-label` value adds some readable text to the equation.

```
<!-- aria-label example -->
<button aria-label="undo">&#xE000;</button>
```

"Button: Undo"

aria-labelledby

In this example, some descriptive text is used to help people understand the use of the button. Since the undo button is referred to in this text, we can code the relationship between the button and its description in an accessible way. This is done via the `id` of the descriptive text.

38. http://en.wikipedia.org/wiki/Private_Use_Areas

```
<p>To go back a step, press the <strong id="undo-text">undo
</strong>button</p>
<button aria-labelledby="undo-text">&#xE000;</button>
```

Accessible relationships will feature more heavily later in the book. For now, be aware that using `aria-label` and `aria-labelledby` is remedial: only do it if you have no other choice. Otherwise, an attractive button labeled with some nested text is preferable.

THE WORDS

So much of accessibility, as well as usability, is about convention. In fact, the backwards arrow originally employed to communicate the purpose of our button is only successful because it is symbolic: its meaning comes from the convention that going left signifies retracting an action.

When you think about it, there are probably better ways to signify retracting an action (especially when you consider that some languages read right-to-left), but the left-pointing arrow is a prevalent convention and — in lieu of a more universal symbol — one not to be sniffed at. We've agreed that's what it means to us.

> *Leave 'creativity' to the bad designers — This is not the place to do something different. If a convention exists, use it.*
> — *Mark Boulton*[39]

Words, like pictograms[40], can also be symbolic and using conventional words for common actions may not be creative but it's usability gold because it creates less cognitive strain[41] for the user.

- *Save*, not *Store*
- *Delete*, not *Destroy*
- *Edit*, not *Permutate*

Send
Let's do this!
"Okay, I will"
"No. Stop trying to sound cool"

39. http://www.markboulton.co.uk/journal/icons-symbols-and-a-semiotic-web
40. http://en.wikipedia.org/wiki/Pictogram
41. http://www.nngroup.com/articles/navigation-cognitive-strain/

TEST.CSS

It's time to add another rule to our *test.css* file. This declaration block uses an array of selectors to determine whether you've at least used some kind of technique to label buttons in your webpage in an accessible way.

```
a:empty:not([aria-label]):not([aria-labelledby]):after,
button:empty:not([aria-label]):not([aria-labelledby]):after,
button:not([aria-label]):not([aria-labelledby])
img:only-child:not([alt]):after,
a:not([aria-label]):not([aria-labelledby])
img:only-child:not([alt]):after {
        background: red;
        color: white;
        content: 'You are not providing enough information about
        what this button does. Please include some text within
        the button.';
}
```

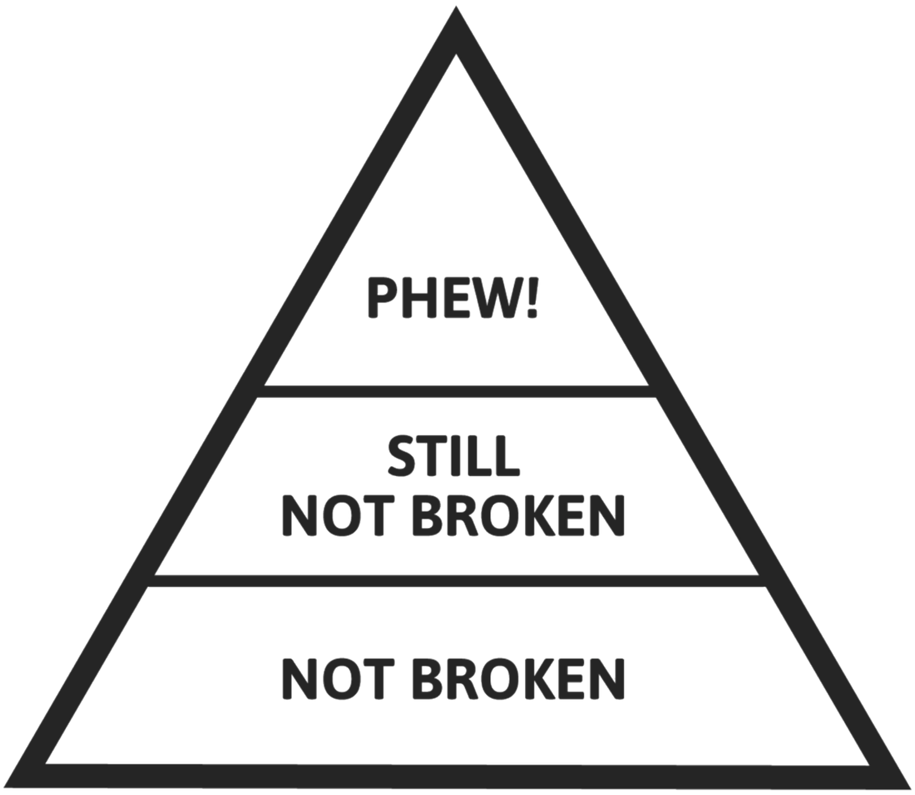
Breaking it down, we are saying, "If the button has no content at all *and* also has no accessible ARIA label, display a warning." In conjunction with that, we are also saying, "If there is some content inside the button and it's just an image *and* that image doesn't have an `alt` attibute, display a warning." Ugly as that CSS looks, it may help you keep your labels in check. ❧

CHAPTER 3:

The WAI Forward

In the last chapter, we established the button as the standard control for web applications. We explored `<button>` as an exemplar of semantic HTML, using it to establish predictable behaviors and displaying it unambiguously, heeding the Web Accessibility Initiative's guidelines.

Even when we branched out to do something a little more radical with CSS transitions, we made sure our technique was a progressive enhancement, built on robust foundations. This way, older browsers and other technologies still had something to fall back on.



As you read through the WCAG 2.0 guidelines, you'll notice they pertain to things you *shouldn't undo* as well as things you should do. Arguably, they're more about not making things inaccessible than making things accessible. For instance, guideline 2.3 implores you not to "design content in a way that is known to cause seizures". It's possible to design a webpage that actively sets out to cause seizures, but you'd have to go out of your way to make something as absurdly irresponsible as that.

Because the W3C's mission was to make the web accessible from the outset, many accessibility features are built in. As responsible designers, it is our job to create compelling web experiences without disrupting the inclusive features of a simpler design. As Scott Jehl puts it:

> *Accessibility is not something we add to a website, but something we start with and risk losing with each enhancement. It is to be retained.*
> *— Scott Jehl (Twitter, Dec 12, 2013)*[42]

Unfortunately, not all websites are destined to be as simple as the provocative manifesto that is "This is a motherfucking website[43]".

As we together embrace the advancements of the web and our newfound power to construct hitherto impossible web-based software, we need to tackle the accessibility of new idioms. We need to find a way to adopt new tools and techniques to "keep the playing field level", as Ron McCallum requested in the first chapter, and maintain a parity of experience between our different users.

It's time to embrace change.

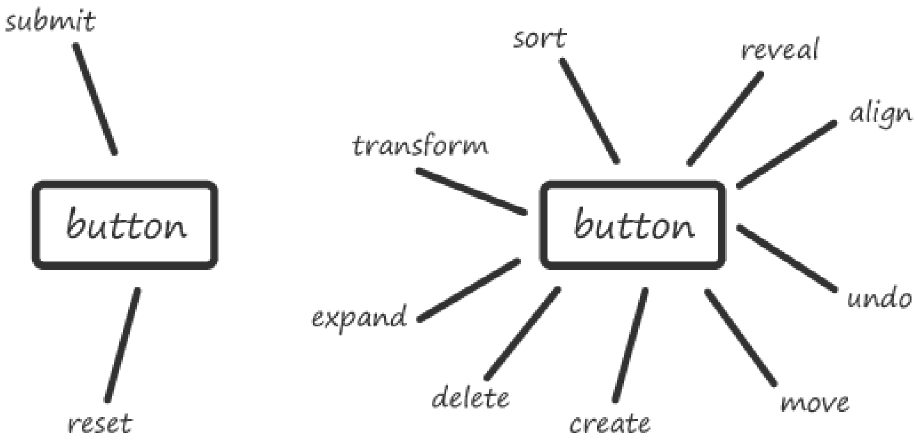
ARIA: A Passion For Parity

WAI-ARIA[44] is an accessibility resource like WCAG 2.0, with certain notable differences. If it helps, you could think of the two resources as siblings: each has been brought up in the same environment and had the same basic values instilled in them, but they differ in personality.

42. https://twitter.com/scottjehl/status/411237303579721728
43. http://motherfuckingwebsite.com/
44. http://www.w3.org/WAI/intro/aria

WCAG 2.0 is the cautious homebody who keeps the home fires burning, while the more gregarious WAI-ARIA has ambitions to take accessibility to new territories.

Unlike WCAG 2.0, ARIA is not only a set of recommendations but a suite of attributes to be included in your HTML. It gives you the tools to alter and increase the amount of information shared about your HTML to users of assistive technologies. When making web apps, this is extremely useful because the roles, properties, states, and relationships of your elements are liable to be a lot more complex and dynamic. One way of looking at it is this: ARIA gives you the tools to meet WCAG requirements in web apps.

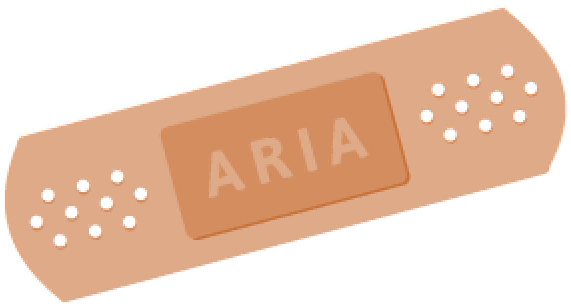


THE TWO PURPOSES OF ARIA

ARIA gives you the ability to reclassify and otherwise augment the perceived meaning (or semantics) of your HTML. That's pretty powerful, but what is the purpose of it? There are two main applications of ARIA.

Remedy

ARIA can be used as a remedy to improve the information provided to assistive technology by poorly coded, unsemantic markup.



For example, a developer might use a `<div>` and some JavaScript to emulate a `type="checkbox"`. They probably shouldn't, but they might. To make this `<div>` actually understandable as a checkbox, the ARIA role of `checkbox`[45] can be added as an attribute, making screen readers think it is, in fact, a standard checkbox. In addition, our developer must use the `aria-checked` attribute to indicate whether the checkbox is indeed checked.

```
<div class="toggle-thingy" role="checkbox" aria-checked="false"
tabindex="0">Yes?</div>
```

It's better to use the proper `input` element, `type` attribute, and `checked` attribute to communicate this information — it is better supported than ARIA (which is relatively modern) and the `input` would also be automatically focusable, like the semantic `<button>`s of the previous chapter (no need for the `tabindex`). Nonetheless, we can employ ARIA in this way to make quick fixes to markup, often without disturbing the design of the application.

Enhancement

As we've established, web applications are more complex than simple web documents and our use of HTML elements tends to exceed the basic semantics gifted to them. ARIA's killer feature is its ability to help authors like you and me communicate many of these more ambitious uses in an accessible way.

Take ARIA's `aria-haspopup` attribute[46]. This represents a property of certain elements which have a hidden submenu. The owner of this property is likely to be an `<a>` or a `<button>` and, without this special attribute, that's *all* they'd be to a screen reader user: no clue would be given that the submenu existed.

```
<li>
    <a href="#submenu" aria-haspopup="true"aria-controls=
    "submenu1">Main link</a>
    <ul id="submenu">
        <li><a href="/somehwere/">Submenu link</a></li>
        <li><a href="/somehwere/else/">Another link</a></li>
    </ul>
</li>
```

As we shall address in the following chapter, "Getting Around", some of these ARIA attributes are expected to be replaced by simple HTML elements and attributes. As I write, the `<dialog>` element is being slowly

45. https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Techniques/Using_the_checkbox_role
46. http://www.w3.org/TR/wai-aria/states_and_properties#aria-haspopup

adopted[47] as the successor of the `dialog` and `alertdialog` ARIA role attributes, for example.

Where possible, superceding ARIA with plain HTML(5) is good because it can simplify writing the markup and centralizes the W3C's advice. However, many attributes which communicate specific, contextual information like the `aria-haspopup` and `aria-controls`[48] properties in the above code, are unlikely to find many mainstream supporters for inclusion as, perhaps, `haspopup` and `controls`.

As Steve Faulkner points out in "HTML5 and the myth of WAI-ARIA redundance[49]", much of ARIA will continue to exist unbested. It is the unique power of ARIA to bridge the gap between the experiences of sighted and unsighted web users which is the subject of much of this book.

Role-Playing

A lot of my friends and colleagues are keen on tabletop role-playing games which, if you're not familiar with them, are games in which participants play fictional characters embarking on quests and fighting battles in fantastical worlds. Although I'm not a big exponent of these games myself, I've noticed similarities between the way characters are role-played in games and the way HTML elements act within web applications. Accordingly, I'm going to use this role-playing metaphor to explain ARIA's *roles*, *properties*, and *states* in greater detail.

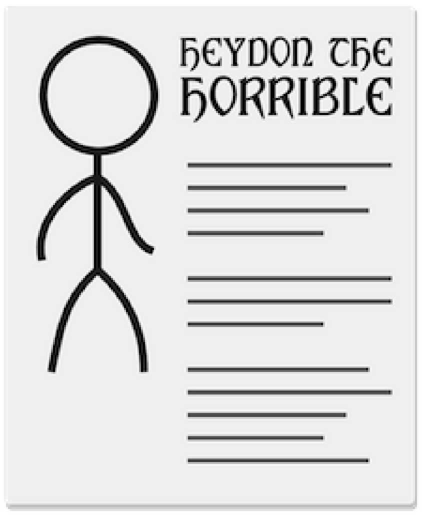
Don't worry, you won't have to be a big role-playing geek to follow along. I'm not!

ROLES

Each player in a role-playing game normally maintains a "character sheet" which lists the important characteristics of the character they are playing. Compared to HTML, the name of the character might be the `id` of an element. Each must be unique.

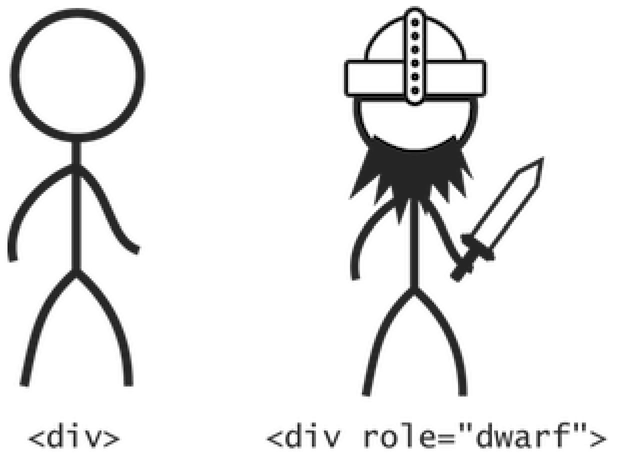
There's a lot more information than that in a character sheet, though. For example, characters dwelling in these fantasy worlds usually belong to one "race" or another. Common standbys are races like elves, dwarves, and trolls. These are like HTML element types: broad groupings of participants with common characteristics.

47. https://twitter.com/stevefaulkner/status/413263499863658496
48. http://msdn.microsoft.com/en-us/library/ie/cc848872%28v=vs.85%29.aspx
49. http://blog.paciellogroup.com/2010/04/html5-and-the-myth-of-wai-aria-redundance/



#heydonTheHorrible

In ARIA, the `role` attribute overrides the element type, much like a role-player overriding their workaday existence as a twenty-first century human in favor of becoming a mighty dwarf. In the example from the last section, an insipid `<div>` assumed the role of a checkbox by having `role="checkbox"`.



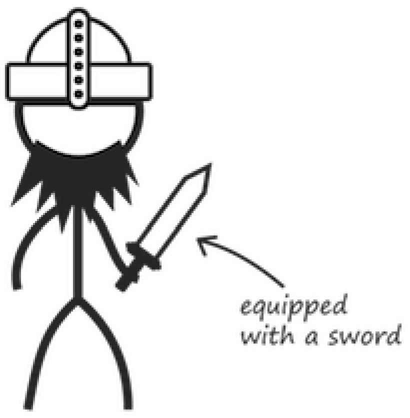
Roles in ARIA[50], like races in role-playing, are a part of the character's identity we might be interested in. We may expect dwarves to be strong and good at constructing machines like we expect `<button>`s to have the characteristics and behaviors we have already discussed. By putting `role="button"` on an element which isn't actually a `<button>`, you are asking assistive technologies to identify it as a button, evoking these characteristics.

50. http://www.w3.org/TR/wai-aria/roles

PROPERTIES

Character sheets with just names and races would be a bit limited. I think we'd all be a bit uncomfortable with so much emphasis on race, too. The whole point of ARIA is that it recognizes elements not just for their generic, reductive classification. Much better to identify characters and elements by their individual assets and talents.

In a typical character sheet, the character will have a set of characteristics listed which, in one way or another, the game will identify as having a certain currency and importance. For instance, you may be an elf, but one who is special for her ability to cast certain magic spells. In much the same way, we looked at an `<a>` element which was made special for having the property that it secreted a submenu. This would be identified to the accessibility layer, just like the basic role, via the `aria-haspopup="true"` property attribute.



`<div role="dwarf" aria-hasSword="true">`

There are a huge number of properties[51] specified and documented. Some are global, meaning any element can have them, while others are reserved for particular contexts and elements. Dwarves are usually precluded from having good longbow accuracy, but it is a common property of elves. The `aria-label` we used to label our button in the previous chapter is global, while `aria-required` — which denotes a required user entry — should only normally be used on form fields or elements with form field roles such as `listbox` or `textbox`.

STATES

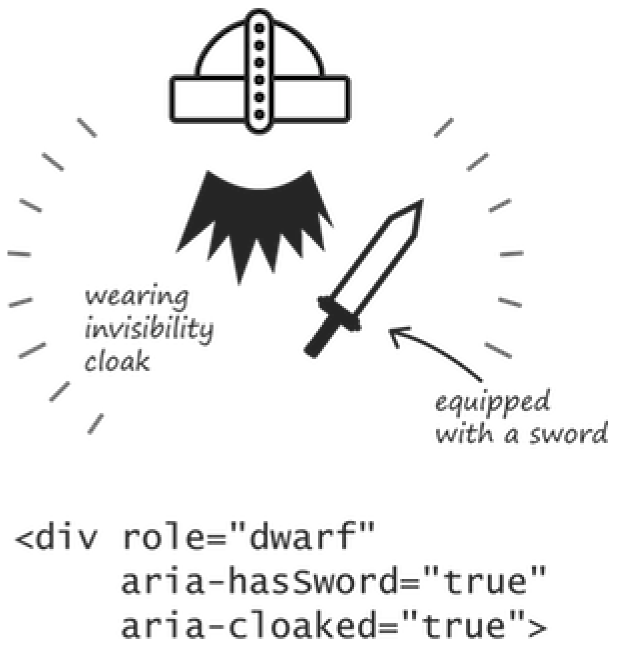
Perhaps the most important distinction between a static web document and an application is that the elements in an application tend to change — sometimes dramatically — according to user interaction and timed events. Depending on what's happening in an application at any

51. http://www.w3.org/TR/wai-aria/states_and_properties

one time, the elements therein could be said to be in certain, often temporary, states.

In role-playing games, you have to keep tabs on the state of your character: how healthy are you? What items have you collected? Who have you befriended? This is all scribbled down, erased, and scribbled down some more on the character sheet. Keeping track of the state of interactive elements is important for accessibility too.

In your application, the state of elements is usually represented visually. In role-playing games and in screen reader buffers this is not the case: you can only imagine. If your dwarf character has donned his magic cloak of invisibility it's probably best to write this down on the character sheet so you remember. Similarly, writing the `aria-hidden` attribute[52] on an element ensures the accessible state of invisibility is recorded properly.



```
<div role="dwarf"
     aria-hasSword="true"
     aria-cloaked="true">
```

States like `aria-expanded`[53], which will feature in the building of collapsible content in chapter 5, "Peekaboo", are announced according to a value of true or false. An item with `aria-expanded="false"` is announced by the JAWS and NVDA Windows screen readers as "collapsed". If — or rather when — it is set to `aria-expanded="true"`, the item is announced as "expanded."

52. http://www.w3.org/TR/wai-aria/states_and_properties#aria-hidden
53. http://www.w3.org/TR/wai-aria/states_and_properties#aria-expanded

Our First ARIA Widget

It's about time we put everything we've learned about roles, properties, and states into practice and build our first ARIA widget.

The term *widget* is often used in JavaScript development to denote a singular pocket of script-enabled interactive functionality. Mercifully, the ARIA definition corresponds and ARIA widgets can be thought of as JavaScript widgets which have been made accessible using appropriate ARIA attributes.

For the following example, we are going to make a simple toolbar; a group of button controls which allow you to organize some content. In this case, I'll provide controls to sort the content alphabetically and reverse alphabetically. Fortunately, we have access to a W3C guide on authoring ARIA widgets called "General Steps for Building an Accessible Widget with WAI-ARIA[54]" which covers a similar toolbar example.

THE TOOLBAR ROLE

There's no such thing as a `<toolbar>` element in HTML unless you create one as a web component[55]. In any case, because there's no *standard* element for toolbars, we need to include the `toolbar` role on our toolbar's containing (parent) element. This marks the scope of the widget:

```
<div role="toolbar">
    /* toolbar functionality goes here */
</div>
```

(Note: There is a `<menu>` element which takes a `type` of `toolbar` but the element has not been adopted by browsers[56], so is unable to provide the information we need.)

It should be obvious what the toolbar is for in our design by its visual relationship to the content it affects. This won't help to communicate anything aurally, however, so we should provide an accessible name for our toolbar via the now familiar `aria-label` property. That's one role and one property so far.

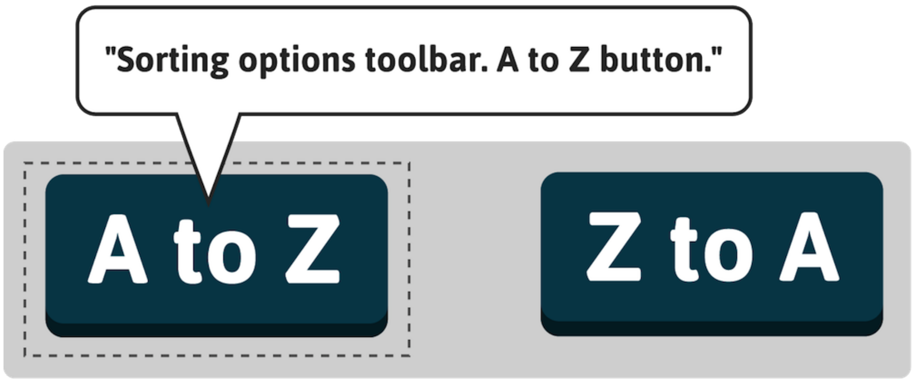
```
<div role="toolbar" aria-label="sorting options">
    /* toolbar functionality goes here */
</div>
```

Now we're going to add the buttons which will be our controls.

54. http://www.w3.org/WAI/PF/aria-practices/#accessiblewidget
55. http://www.techrepublic.com/blog/web-designer/learn-more-about-web-components-with-these-demos/
56. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/menu

```
<div role="toolbar" aria-label="sorting options">
    <button>A to Z</button>
    <button>Z to A</button>
</div>
```

Even without the additional widget properties and states, we've already improved the user's recognition of the toolbar: using the NVDA screen reader or the JAWS screen reader with Firefox, when a user focuses the first button, they are informed that they're inside a toolbar and — thanks to the `aria-label` — what it is for.



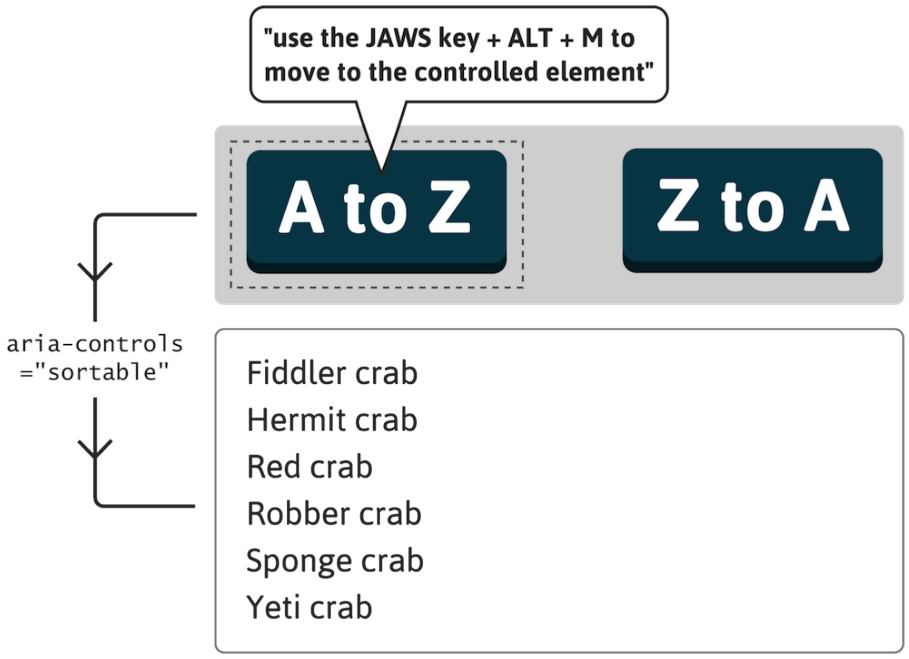
THE RELATIONSHIP

So far, we haven't actually connected our toolbar to the content it controls. We need a relationship attribute, which is a special kind of property attribute communicating a relationship between elements. Our widget is used to control the content, to manipulate and reorganize it, so we're going with `aria-controls`. We join up the dots using an `id` value, just as we did in the earlier popup menu example.

```
<div role="toolbar" aria-label="sorting options"
aria-controls="sortable">
    <button>A to Z</button>
    <button>Z to A</button>
</div>

<ul id="sortable">
    <li>Fiddler crab</li>
    <li>Hermit crab</li>
    <li>Red crab</li>
    <li>Robber crab</li>
    <li>Sponge crab</li>
    <li>Yeti crab</li>
</ul>
```

Note how we've added `aria-controls` to the toolbar itself, not each individual button. Both would be acceptable but using it just the once is good for brevity and the buttons should each be considered individual components belonging to the controlling toolbar. To check which properties and states are supported for widget roles like `toolbar`, the specification provides a list of "inherited states and properties[57]" in each case. You are advised to consult this when building a widget. As you will see[58], `aria-controls` is an inherited property of `toolbar`.

Some screen readers do very little explicitly with this relationship information, while others are quite outspoken. JAWS actually announces a keyboard command to let you move focus to the controlled element: "Use *JAWS key* + *ALT* + *M* to move to controlled element." Once you've affected it, you might want to go and inspect it, so this is JAWS helping you to do just that.


"use the JAWS key + ALT + M to move to the controlled element"
A to Z
Z to A
aria-controls ="sortable"
Fiddler crab
Hermit crab
Red crab
Robber crab
Sponge crab
Yeti crab

PRESSED AND UNPRESSED

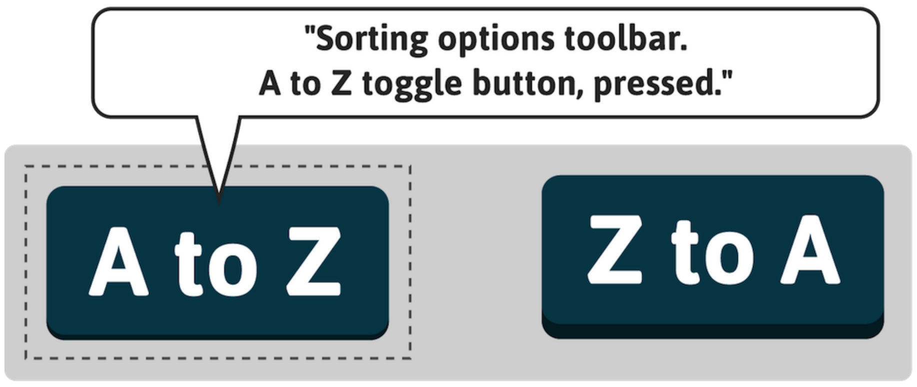
Depending on which sorting option is our current preference, it could be said that the button which commands that option is in a *selected* or *pressed* state. This is where the `aria-pressed` state attribute comes in, taking a value of `true` for pressed or `false` for unpressed. As we covered, states are dynamic and should be toggled with JavaScript. On loading the page, just the first button will be set to `true`.

57. http://www.w3.org/WAI/PF/aria/roles#toolbar
58. http://www.w3.org/WAI/PF/aria/roles#toolbar

```
<div role="toolbar" aria-label="sorting options"
aria-controls="sortable">
    <button aria-pressed="true">A to Z</button>
    <button aria-pressed="false">Z to A</button>
</div>
<ul id="sortable">
    <li>Fiddler crab</li>
    <li>Hermit crab</li>
    <li>Red crab</li>
    <li>Robber crab</li>
    <li>Sponge crab</li>
    <li>Yeti crab</li>
</ul>
```

It's good practice to pair the styles for active (`:active`) buttons we created in the last chapter with the styles for `aria-pressed` buttons. Each, whether momentarily or semi-permanently, are buttons which have been *pushed down*.

```
button:active, button[aria-pressed="true"] {
    position: relative;
    top: 3px; /* 3px drop */
    box-shadow: 0 1px 0 #222; /* less by 3px */
}
```



button visually depressed to denote state

When focusing a button with `aria-pressed` present using either NVDA or JAWS with Firefox, the button is identified as a "toggle button". Using the latest version of JAWS and focusing a button with `aria-pressed="true"` will append the word "pressed" to the announcement accordingly. In the ChromeVox[59] screen reader for the Chrome

59. http://www.chromevox.com/

browser, an `aria-pressed="true"` button is announced as "button pressed" and `aria-pressed="false"` as "button not pressed". To a greater or lesser extent, most modern browsers and screen readers articulate clarifying information about the state or potential state of these buttons.

KEYBOARD CONTROLS

Not quite there yet. For toolbars – as with many ARIA widgets – the W3C recommends certain keyboard navigation features[60], often to emulate equivalent desktop software. Pressing the left and right arrow keys should switch focus between the buttons and pressing *Tab* should move focus out of the toolbar. We'll add `tabindex="-1"` to the list and focus it using JavaScript whenever the user presses *Tab*. We do this to allow users to move directly to the list once their sorting option is chosen. In a toolbar with several buttons, this would save them from potentially having to tab through a number of adjacent buttons to get to the list.

```
<div role="toolbar" aria-label="sorting options"
aria-controls="sortable">
        <button aria-pressed="true">A to Z</button>
        <button aria-pressed="false">Z to A</button>
    </div>
    <ul id="sortable" tabindex="-1">
        <li>Fiddler crab</li>
        <li>Hermit crab</li>
        <li>Red crab</li>
        <li>Robber crab</li>
        <li>Sponge crab</li>
        <li>Yeti crab</li>
    </ul>
```

```
$(listToSort).focus();
```

We'll talk in more depth about managing focus like this in later examples.

ALL DONE

That concludes our first ARIA widget. As with many of the examples in this guide, I have made a live demo[61] available to play with and test. Re-

60. http://www.w3.org/WAI/PF/aria-practices/#toolbar

member: It's not really about sorting, per se; that's all done with JavaScript. The aim here is to make explicit the relationships and states of our application, so that whatever we are doing to our content — sorting it, editing it, searching it, creating it, recreating it — our keyboard and screen reader users are kept in the loop.

Know The Rules

As we progress through this book, making increasingly complex ARIA widgets, it would be impractical to test every single little tweak and nuance manually in all of the browser and screen reader combinations. Although testing is essential, avoiding basic screw-ups in the first instance needs little more than a handful of guiding principles to keep close at hand.

To start out on the right foot, we are going to abide by *the three rules of ARIA use*[62] as laid out by the the WAI-ARIA experts at the W3C. Let's study these now or else we might do something embarrassing.

FIRST RULE OF ARIA USE

> *If you can use a native HTML element [HTML5] or attribute with the semantics and behaviour you require* **already built in**, *instead of re-purposing an element and adding an ARIA role, state or property to make it accessible,* **then do so**.

Remember that ARIA is used to remedy or enhance HTML code. There's no need to use ARIA attributes if the built-in or native semantics of HTML will do. As in the checkbox example earlier on, the use of the `checkbox` role was suboptimal and would only come about under certain circumstances where the inert `<div>` element could not be easily replaced.

In some cases, the support for new HTML attributes is currently less widespread than the support for their ARIA counterparts. For example, tests from a couple of years back[63] showed that `required` was not supported at all across screen reader and browser combinations, while support for `aria-required` was strong. For the time being, you should take a belt-and-braces approach by including both attributes to **maximize compatibility**: a phrase we'll hear more of as we progress.

61. http://heydonworks.com/practical_aria_examples/#toolbar-widget
62. http://www.w3.org/TR/aria-in-html/#first-rule-of-aria-use
63. http://wps.pearsoned.com/wps/media/objects/8956/9171771/aria-required.html

```
<input type="text" id="text-entry" required aria-required="true"
/>
```

SECOND RULE OF ARIA USE

Do not change native semantics, unless you really have to.

As the same "Using WAI-ARIA in HTML" guide explains[64], adding a `role` attribute to an element will override the native semantics of that element. So, adding `checkbox` to `<div>`, helpfully makes that unsemantic `<div>` parade as a checkbox. Less helpfully, adding a role of `button` to an `<h1>` would effectively make that `<h1>` a button – at least, in the accessibility layer where it would be announced. So, the information about the element being a first-level heading would be lost.

When we make clickable headings later on in "Peekaboo", chapter 5, we instead use JavaScript to place a `<button>` as a child of the heading, meaning we keep the heading semantics, which are still important. It is precisely this which the W3C recommends:

```
<h1><button>heading button</button></h1>
```

THIRD RULE OF ARIA USE

All interactive ARIA controls must be usable with the keyboard.

Sound familiar? That's because it echoes WCAG 2.0's guideline, 2.1, "Make all functionality available from a keyboard". And, with that, we've come full circle.

In our toolbar example, basic keyboard navigation is made possible simply by including focusable `<button>`s as controls. We enhanced this by enabling users to switch between buttons using the arrow keys. We shall explore more complex custom keyboard functionality using programmatic focus management in later examples.

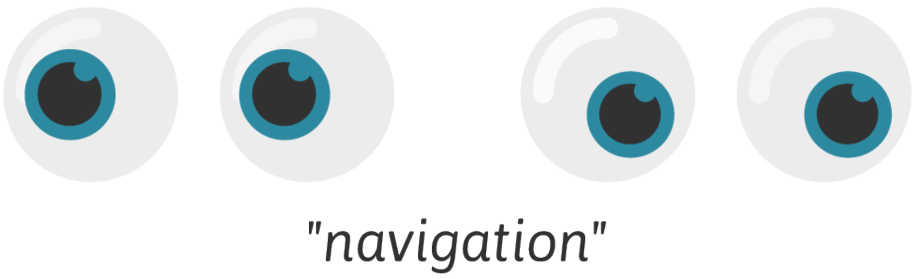
That is all to come. First, we shall put our application functionality to one side and be addressing how to actually *find* that application functionality in the first place. Mobility is a big part of accessibility online and off, so we'll look at different ways to help our users get around. ❧

64. http://www.w3.org/TR/aria-in-html/#what-does-adding-a-role-do-to-the-native-semantics

CHAPTER 4:

Getting Around

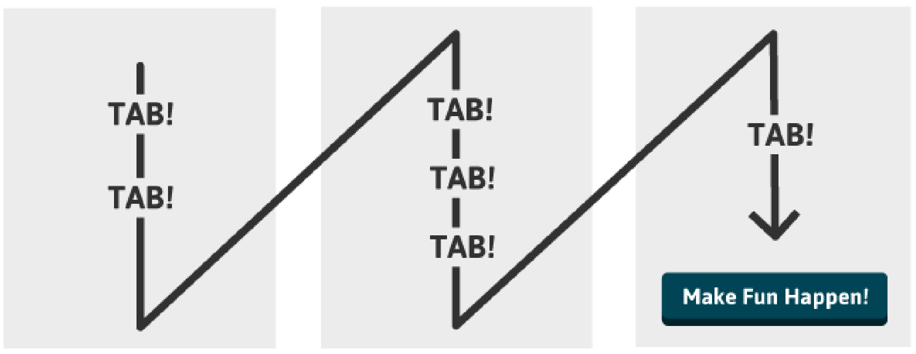
Contemplate the ease with which I can quickly consume an interface. Darting my eyes back and forth over the screen, it takes moments for me to chart its topology. My apprehension of how much information is there, which parts of it are important and which bits I'd really rather avoid is almost instantaneous. Can't see it all at once? I simply grab the scrollbar and briefly dip below the so-called fold. "OK, that's a lot of stuff. Maybe I'll bookmark it for later."



Hyperlinks, which let us traverse between pages are innately accessible and have been from the beginning. They can be focused and are identified by assistive technologies for what they are: "Link." The same `<a>` element can also take you to parts of the same page and most screen readers will helpfully announce focused links on the same page as, well, "same page link."

So that's getting around done, then. Except it isn't.

The structure of webpages which many people consume visually, set to a grid, is no structure at all to those who are not looking. To screen reader users, *grids do not exist*. The extra time it takes the page to load your perfect golden ratio grid system framework is all for nothing to them. Even keyboard users (who can at least adjust their gaze to a new item) are not really helped by a multiple column layout. They still have to cover the same distance, just sideways.



While I can scope a webpage with little more than a cursory glance, if the developer of the page hasn't been thoughtful, keyboard and screen reader users are forced to trudge through a formless no man's land of text, links and little more.

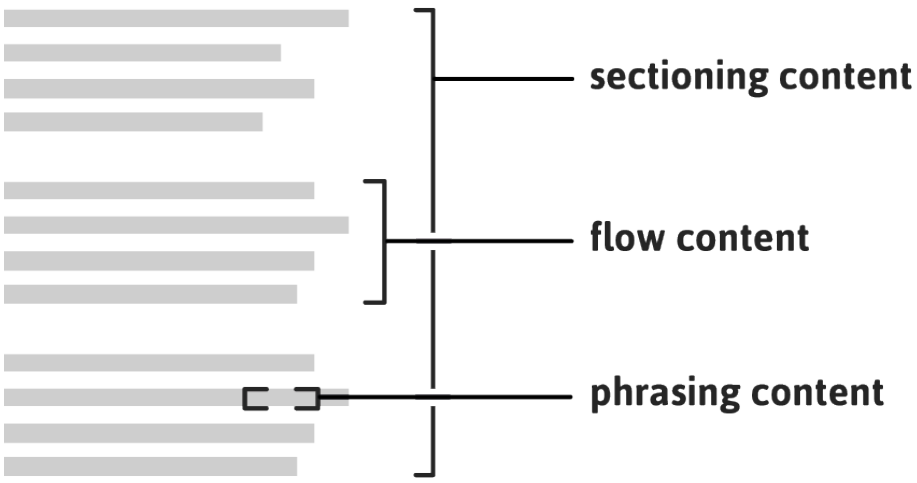
Like any problem, this needs to be broken down. In this case literally, since we need to break the page down into manageable and recognizable regions. Using semantic HTML enhanced with a few choice ARIA attributes, we can begin to draw our users an invisible map of our app. Once they know where things are, we can help our users get between them.

Dividing Things Up

In HTML5 we were given a handful of new elements. These sectioning elements[65] (`<section>`, `<article>`, `<nav>`, and `<aside>`) are no doubt familiar to you already, but I wonder whether you've considered them in terms of accessibility?

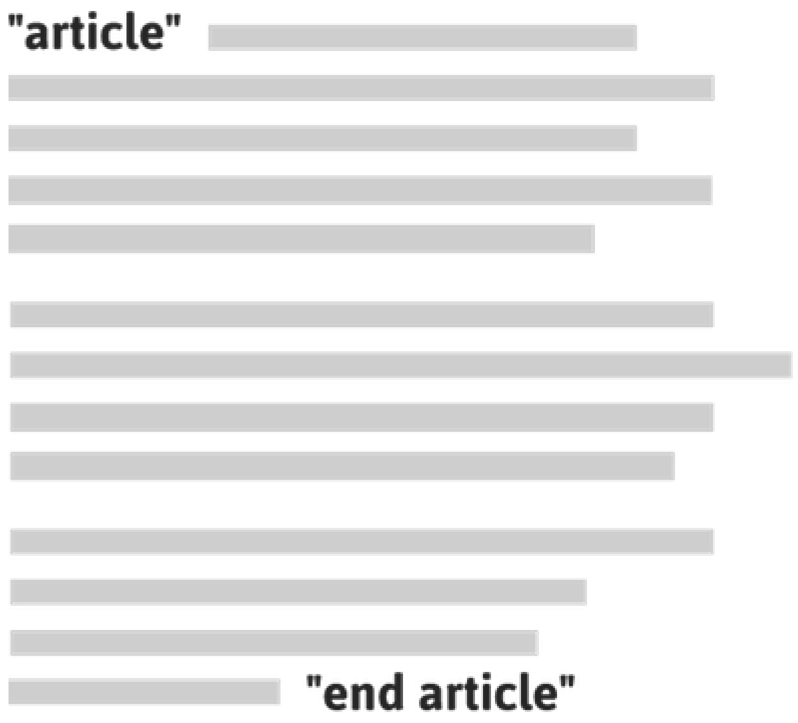
Sectioning content is one of three basic content types in HTML. You have phrasing content (inline elements like `<em>` and `<strong>`); flow content (mostly block-level elements like `<p>` and `<div>`); and sections which are used to define the scope of parts of your page. In other words, you have small bits of content, larger bits of content, and big areas of scoped content.

In well-structured webpages, each piece of phrasing content should belong to a piece of flow content, which should probably belong to a section. If no HTML5 sectioning elements are used, `<body>` can be thought as the single section that wraps the lot.



65. http://www.w3.org/WAI/GL/wiki/Using_HTML5_section_elements

The potential advantage of using HTML5's sectioning elements is they can break up content into manageable chunks. When using some versions of the screen reader JAWS, `<section>`s and `<article>`s are announced as such when you reach them in the page, then concluded (with "Article end," for instance) when you have reached their end.



The superiority of using a sectioning element over the previously common `<div>` should be clear: while sectioning elements are often identified aurally, `<div>`s never are. The `<div>` may be good for constructing grids but it's really just masonry. It doesn't provide any meaning to users of AT. In fact, it isn't supposed to impart any semantic information at all.

> *In HTML, the* `span` *and* `div` *elements are used for generic organizational or stylistic applications, typically when extant meaningful elements have exhausted their purpose.*
> — *Wikipedia*[66]

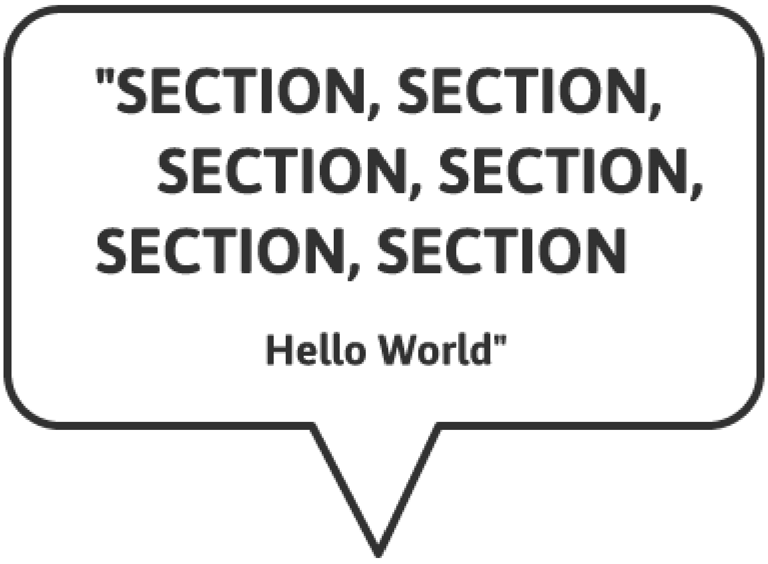
The potential advantage of using sectioning elements breaks down when `<section>`s are wrongly adopted as an equivalent to `<div>`s and you see examples of markup like this:

```
<section class="outer">
    <section class="inner">
        <section class="content">
```

66. http://en.wikipedia.org/wiki/Span_and_div

```
            <!-- content goes here -->
        </section>
    </section>
</section>
```

It is considered bad practice to nest `<div>`s too deeply because it adds unnecessary bloat to your HTML code, but nesting `<section>` creates much worse problems. At least not each and every extraneous `<div>` is announced to the quickly tiring screen reader user!



For this reason, the HTML5.1 specification[67] — the newest version of the HTML5 specification — has recently been altered to strongly encourage the use of headings to label individual sections and emphasize their true purpose.

> *The theme of each* `section` *should be identified, typically by including a heading (*`h1`*–*`h6` *element) as a child of the* `section` *element.*

In addition, the W3C now recommends that browser and assistive technology vendors "only convey the presence of [...] `section` elements when the `section` element has an accessible name[68]". Anticipating some author error, we would hope that this safeguard becomes widely adopted.

We shall talk about the role of headings in webpage accessibility later. For now, suffice it to say that users of the NVDA and JAWS screen readers can use keyboard shortcuts to navigate between headings, which essentially means they can jump between sections. For instance,

67. http://www.w3.org/html/wg/drafts/html/master/sections.html#the-section-element
68. http://www.w3.org/html/wg/drafts/html/master/dom.html#sec-implicit-aria-semantics

in JAWS the keys 1–6 correspond to the HTML heading elements `<h1>` to `<h6>`. To give you a head start on using JAWS keyboard shortcuts or "quick keys", WebAIM has a useful guide[69].

TEST.CSS

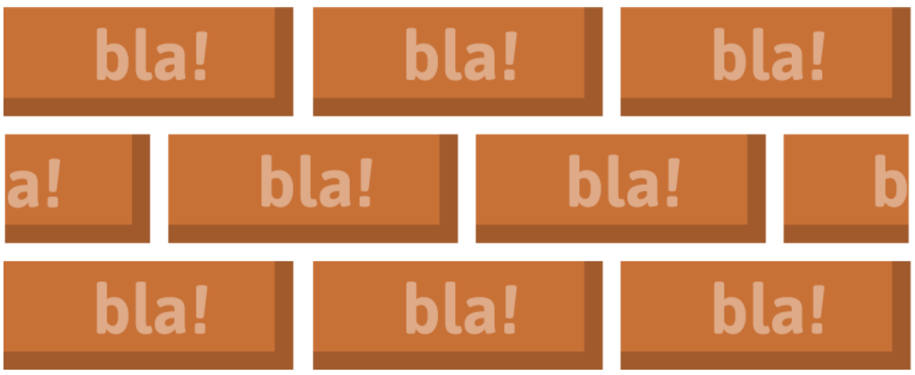
To check that we haven't become overzealous with our `<section>`s and started using them like `<div>`s, we can include a fairly simple rule to check. Add this to your *test.css* style sheet.

```
section > section:first-child:after {
    background: red;
    color: #fff;
    content: 'Warning: it looks like you are using sections like
    divs. Sections should each have a heading';
}
```

Famous Landmarks

Even when used sparingly and correctly, sectioning elements can only improve the way users read content in a linear fashion, from top to bottom. It is, as stated, only their headings which allow users to jump between them and skip the boring parts. Sectioning elements are like building blocks: helpful in terms of construction but little more than generic pieces, just slightly better than old fashioned `<div>`s.

What we really need is a way to identify the major components that make up just about any well-coded webpage; famous landmarks, if you will, in the design of an ordinary page. This is where ARIA's landmark roles[70] step in. Building a webpage from sectioning elements is like laying bricks in a wall. Working with landmarks is more like identifying the essential organs that make up the page's anatomy.



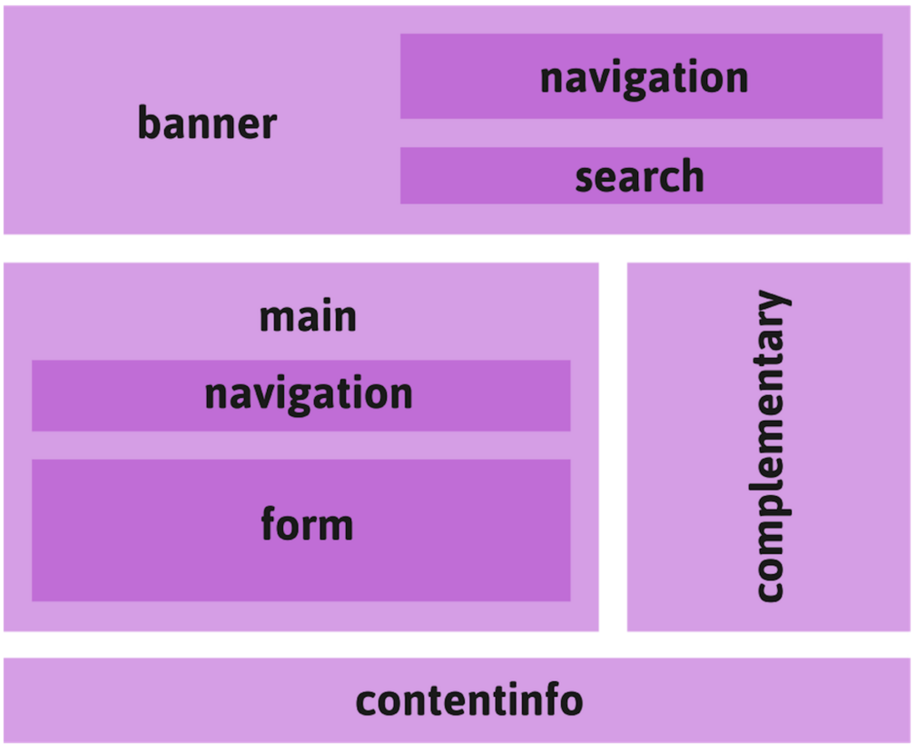
69. http://webaim.org/articles/jaws/#quick
70. http://www.nomensa.com/blog/2010/wai-aria-document-landmark-roles/

Landmarks relate to WCAG's guideline 2.4[71]: "Provide ways to help users navigate, find content, and determine where they are." As we shall explore, landmarks can be particularly helpful in the design of web application interfaces because they can be used to quickly move between key parts of the application.

The landmark roles:

- **banner** (`role="banner"`)
- **contentinfo** (`role="contentinfo"`)
- **main** (`role="main"`)
- **navigation** (`role="navigation"`)
- **complementary** (`role="complementary`)
- **search** (`role="search"`)
- **form** (`role="form"`)

Let's imagine a page which, thanks to a conscientious designer, includes each of these landmarks...



Now let's revisit our landmarks list with some clarification on what the roles are for.

71. http://www.w3.org/TR/WCAG20/#navigation-mechanisms

- **banner**: the preamble to the page, usually containing the main `<h1>` heading and sometimes a `role="navigation"` landmark, too. Can only be used once per page.
- **contentinfo**: information about the page and the website as a whole. The best place to put copyright and contact information. Can only be used once per page.
- **main**: this is where the main content of your page goes. Visitors who've already read your banner on another page may want to skip straight to this. Can only be used once per page.
- **navigation**: a landmark containing links to other pages of your site or important sections of the page itself. You can have multiple `role="navigation"` blocks. Can be nested in other landmarks such as banner and main. Can be used more than once per page but exercise restraint: it is not to be used on just any list of links.
- **complementary**: easy to spell incorrectly as *complimentary*. This constitutes a landmark to contain subsidiary or tangential information. In HTML4 we might have labeled a prototype for this landmark *sidebar*. That doesn't mean it has to appear on the side of your page, though: fat footers containing Twitter updates, lists of links to recommended sites, and other content probably qualify as complementary. This role maps to HTML5's `<aside>` element. Probably best used only once per page.
- **search**: a special role for any form in the page that lets you search or filter the site's content. You may want to use the `search` role for site-wide searches, and searches within sections or single pages of a site. In which case, more than one of these may be applicable per page.
- **form**: a generic role related to `<form>` used to identify any important areas for user input on the page. In single page applications, this role would identify any interactive parts of the application.

TELEPORTATION

ARIA landmark roles, like other ARIA attributes, are gleaned from the HTML and communicated, via the web browser, to a screen reader to reveal special information about the page's structure.

Like headings, landmarks can be used as navigational aids because screen reader vendors like JAWS and NVDA provide keyboard shortcuts to move between them. For example, in JAWS 15 you can press the *R* key to go to the next landmark or *Shift* + *R* to go back one.

Compared to hopping from one link to another, this is like teleporting across continents. Better still, both JAWS and NVDA provide special dialogs which list the landmarks in the page and let you move to them.

In NVDA, you access this dialog by pressing *Insert* + *F7*. Helpfully, landmarks nested within other landmarks are indicated as being so placed:

- banner
  - navigation
  - search
- main
  - navigation
  - form
- complementary
- contentinfo

In the above example, note how we have employed two navigation landmarks. This is totally legitimate: by nesting, we are able to define their different roles within the overall structure. In this case, the banner navigation would probably take you to other parts of the site, while the navigation block inside *main* would most likely use same-page links to provide navigation for the page's content.

A Keyboard Tour Of Famous Landmarks

The dialog interface for landmarks is a powerful tool and credit is due to the vendors of NVDA and JAWS for implementing similar solutions, making switching between their respective products nice and easy.

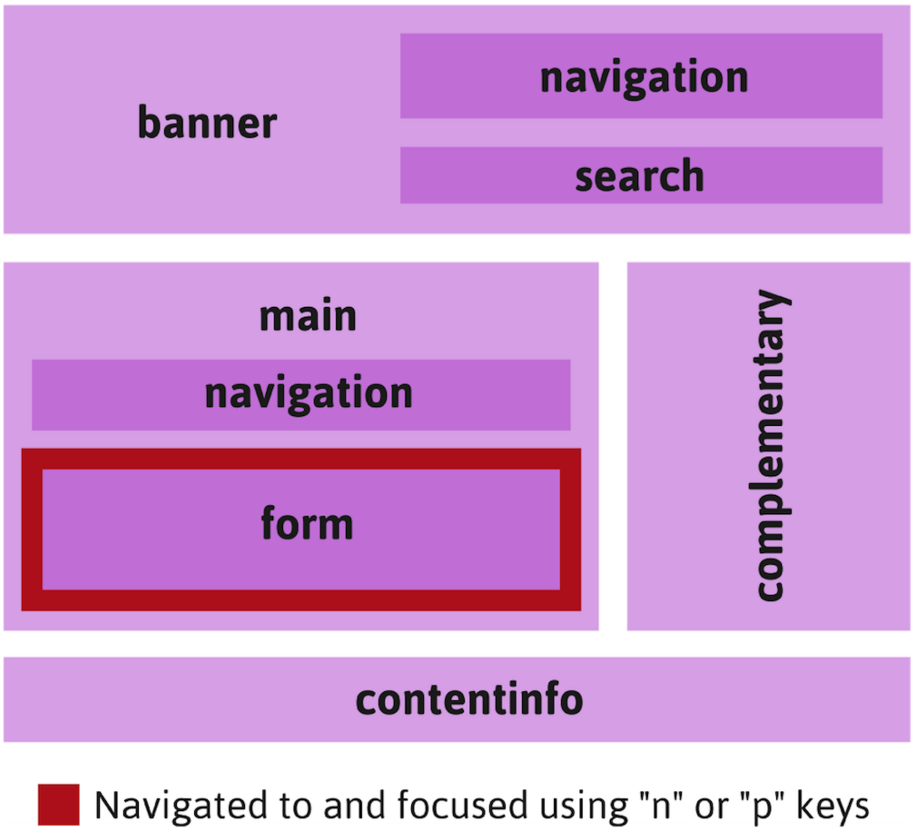
Unfortunately, keyboard users miss out, because the screen reader software must be running to provide this special functionality. Not to worry: someone has thought of this. Users of Firefox are able to install a free extension[72] which provides simple keyboard shortcuts for jumping between landmarks (and implied landmarks — see following section).

- **n** — takes you to the next landmark
- **p** — takes you to the previous landmark

Just as focusing buttons and other controls should be indicated by a visual cue, arriving at a landmark using the extension will highlight that

72. https://github.com/davidtodd/landmarks

landmark with a colored border. The difference is that navigating by landmark happens on a much bigger scale and helps you cover a lot more ground. In applications, where landmarks surrounding the `<main>` area essentially represent tools, this can make performing tasks a lot quicker.



CODING LANDMARKS

ARIA is a bridging technology[73]. It was developed as an extension of HTML to provide accessible semantics lacking in the language and, hopefully, stimulate interest in adopting the ARIA attributes as simple HTML elements and attributes later on. Put another way, ARIA is a prototype for the richer HTML we should like to see in the future.

Sometimes, HTML is able to catch up with ARIA and this can result in some confusion. When do we stop using ARIA and start using simple HTML? There's no easy answer to that, so the best tactic for now is to write ARIA attributes for any HTML elements that would later convey meaning without the help of ARIA.

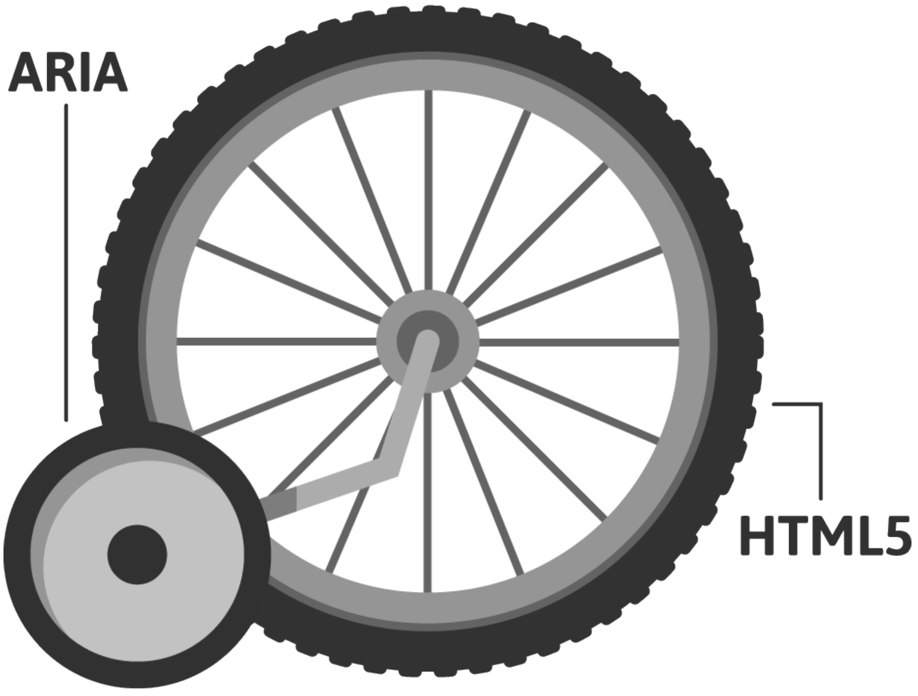
For example, the semantic meaning of the explicit banner role should be considered no different from the meaning conveyed by any `<header>` element that exists as a direct child of `<body>`. Since

73. http://www.w3.org/TR/wai-aria/introduction#co-evolution

<header>s are used to introduce sections and <body> is the biggest section, browsers should be able to interpret this <header> as a banner without extra prompting. Most won't, though — not yet. So, we leave the role in as a helping hand.

```
<body>
    <header role="banner"> <!-- implicit / explicit banner -->
        <h1>Title of page</h1>
        <p>Introductory paragraph</p>
    </header>
    <article>
        <header> <!-- normal header -->
    ... etc...
```

If it helps, think of it like training wheels on a bicycle. You don't remove the bike's wheels just because the training wheels are there: the training wheels are there to support the bike's own wheels until they can be used on their own. The W3C maintains a recommendations table[74] that shows you which HTML elements are equivalent to which ARIA attributes if you need to look up anything.



As Léonie Watson (senior accessibility engineer at The Paciello Group) warns in her "Rock 'n' Roll Guide to HTML5 and ARIA"[75], you should be careful not to use an element and a role separately. In the following in-

74. http://www.w3.org/TR/aria-in-html/#recommendations-table
75. http://www.slideshare.net/LeonieWatson/generate-2013-09

correct example, the `role="navigation"` attribute should be on the `<nav>` element because the ARIA attribute and the element imply the same thing. Separating the role from the `<nav>` creates duplicate indicators for the one landmark. Screen readers would be liable to read "*navigation landmark, navigation landmark*"

```
<nav>
    <ul role="navigation">
        <li><a href="">...</a></li>
        <li><a href="">...</a></li>
        <li><a href="">...</a></li>
    </ul>
</nav>
```

The Main Event

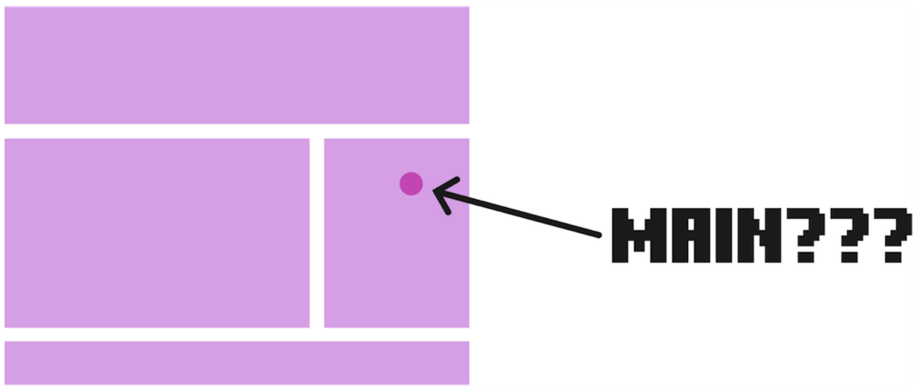
The *main* landmark is rather special in that a unique HTML element, `<main>`, is the direct offspring of the ARIA landmark `role="main"`. The case for marking up the main contents of webpages and applications was so strong that it was agreed an actual tag should take care of the job.

> *The main content area consists of content that is directly related to or expands upon the central topic of a document or central functionality of an application.*
> *– W3C Editor's Draft*[76]

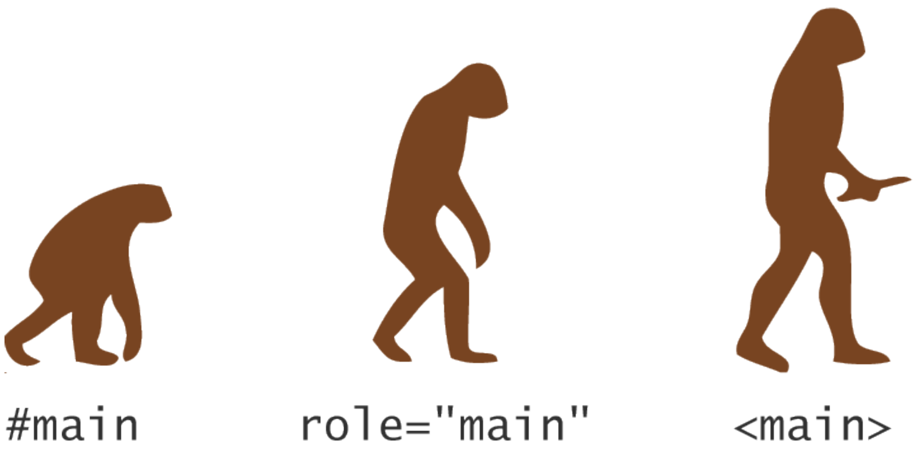
Whether you think of `<main>` as a landmark or everything that's left over after you've set your other landmarks, it quickly became clear that browsers were not clever enough to determine the main part of a page for themselves. A so-called Scooby Doo algorithm[77] for deducing the main content was proposed but early experiments showed it was, well, totally rubbish.

The reason *main* became the responsibility of authors is that HTML, like English and other natural languages, is a way to express meaning, not just procedure. Only human HTML authors truly grasp the individual semantic structure they are putting forth.

76. http://www.w3.org/html/wg/drafts/html/master/grouping-content.html#the-main-element
77. https://willnorris.com/2013/07/scooby-doo-algorithm



It's worth remembering that accessibility is often about empathy, and computers are not inclined to be empathetic: only the humans instructing those machines are capable of that. The `<main>` element is an exemplar of empathetic specification, having taken into account the needs of users and the conventions of authors in its design.



GETTING TO THE JUICY STUFF

A key feature of `<main>` is to signify contents that are unique to the particular page. As the specification puts it, `<main>` "includes content that is unique to that document and excludes content that is repeated across a set of documents".

Keeping in mind that keyboard users and screen readers traverse documents from top to bottom, and the common `role="banner"` landmark usually resides at the top, you can see the problem:

- Users don't need to be told your site's name more than once.
- Users don't want to read your corny slogan more than once (if at all).
- Users especially don't want to have to repeatedly encounter the `<iframe>` banner advert you've lodged up there.

Including a link reading something like “skip to content” right at the top of the page (so it’s the first thing users encounter when they arrive) is a time-honored convention. This should link to the main part of the page as defined similar to below:

```
<main role="main" id="main">...</main>
```

However, it is often coded incorrectly. Because the skip link is only there for keyboard and screen reader users, designers like to tidy it away for users who don’t need it. The first mistake is by using a `display` value of `none`, which hides the link from sighted and screen reader users alike.

```
a.skip {
    display: none; /* nobody can see this */
}
```

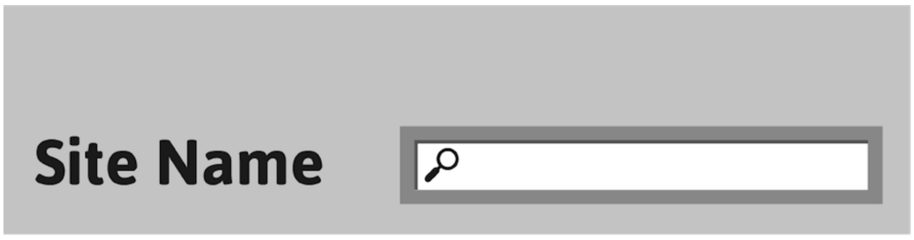
To trick screen readers into thinking the link should be visible, you could simply move it off screen, perhaps using `position: relative` and a negative value like `top: -100px`. The second and more common mistake is to not reveal the skip link to keyboard users. As documented by David Walsh[78], it’s possible to reveal the link to keyboard users when they tab to it by reverting the position value on focus:

```
a.skip:focus {
    top: 0;
}
```

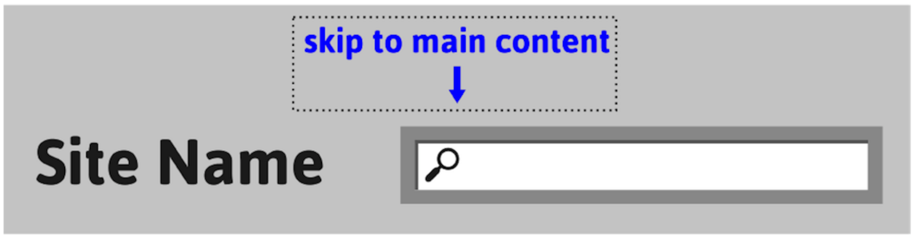
To make revealing the link more obvious, you could even use a CSS3 transition[79] to animate the position and slide the link into view.

```
a.skip {
    position: relative;
    top: -100px;
    transition: position 0.5s ease;
}

a.skip:focus {
    top: 0;
}
```

78. http://davidwalsh.name/accessibility-elements
79. https://developer.mozilla.org/en-US/docs/Web/CSS/transition



a.skip



a.skip:focus

As described, using the landmarks dialogs offered by screen readers or the landmarks extension are more elegant ways to let your visitors get to the important stuff. Nonetheless, only modern setups, sometimes requiring deliberate configuration, will support these features. To help users stuck with older systems, it's often helpful to try some little techniques of our own, like this one.

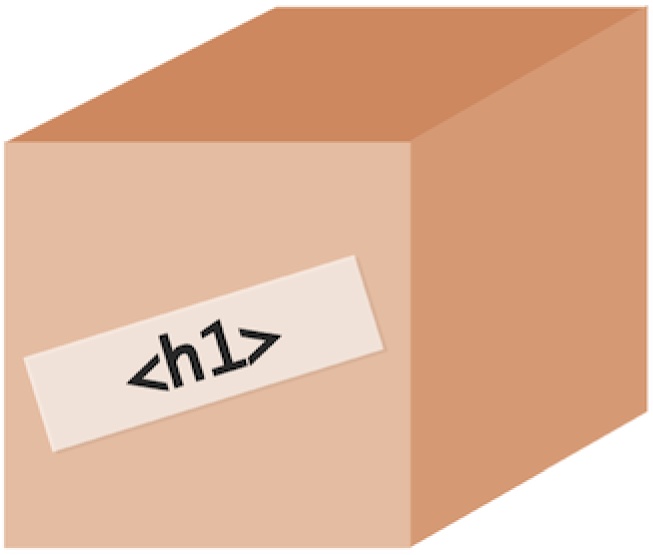
Who knows: if the solutions you come up with really take off, you might see them in a future specification. That's what being part of a community is all about.

Don't Forget Headings!

For many of us, our first encounter with the importance of headings comes with learning their impact on search engine optimization (SEO). It is well known that search engines like Google give more weight to the words and phrases in important headings like `<h1>` and `<h2>` when trying to determine what a site is about. If we have a site that sells guitar amplifiers, we can be proactive and put this important term in our main heading(s). This will enable Google to deem our site relevant to budding guitarists searching with this term.

In accessibility terms, the story is ever so slightly different.

The `<h1>` text is still considered important, but more for its structural significance. When a screen reader announces the `<h1>` heading as "First-level heading: Guitar Amplifiers," the first-level heading qualification indicates that guitar amplifiers are an outer theme to which sub-themes are likely to belong.



Headings are labels for content

As with nesting landmarks, nesting content by heading level involves describing a structure of ownership and telling the user which bits of content belong to which. Nothing is meaningful without context and, since context is dependent on structure, structure surely aids comprehension.

- h1. Stuff I do
  - h2. Play guitar
    - h3. Blues guitar
    - h3. Sludge metal guitar
  - h2. Write
    - h3. Writing about HTML
    - h3. Writing about typography

HEADINGS AND SECTIONS?

For some time it was felt that when we started using sectioning elements to describe structure, we could just use `<h1>` headings everywhere and their level would be calculated algorithmically. Unfortunately, this was utterly useless because it couldn't work with browsers that didn't support sectioning elements. Accordingly, and for the benefit of screen reader users experiencing HTML5 pages, the W3C's advice was changed[80].

To express the above structure with sections *and* headings, the headings for each section should reflect the nesting level or rank of the section itself:

```
<body>
    <h1>Stuff I do</h1>
    <section>
        <h2>Play guitar</h2>
        <section>
            <h3>Blues guitar</h3>
        </section>
        <section>
            <h3>Sludge metal guitar</h3>
        </section>
    </section>
    <section>
        <h2>Write</h2>
        <section>
            <h3>Writing about HTML</h3>
        </section>
        <section>
            <h3>Writing about typography</h3>
        </section>
    </section>
</body>
```

Because structure is paramount, skipping heading levels should be avoided[81]. Remember: `<h4>` doesn't mean the content is fourth most important and not really worth bothering with; it labels a fourth-level subsection — the information within could still be key. Skipping heading levels — like placing an `<h4>` directly after an `<h2>` — makes a nonsense of the structure and leaves some users wondering where they are.

REMEDIAL HEADINGS

As we covered in chapter 3, "The WAI Forward", one of ARIA's useful applications is in remedying the poor accessibility of badly coded webpages.

80. http://lists.w3.org/Archives/Public/public-html/2013Feb/0125.html
81. http://accessibilitytips.com/2008/03/10/avoid-skipping-header-levels/

Let's imagine that a professional designer, Johnny Paycheck, has coded a website with the headings all out of order. The following `<h2>`, for instance, appears inside a `<section>` labeled with an `<h3>`. The mistake he made was to choose headings based on the visual impact of their font size. He just wanted this bit of text to look relatively big and that's all.

```
<h2>Text that looks big</h2>
```

Then let's imagine that another developer, Anna Muggins, has accepted the challenge of making the webpage in question more accessible. She's not considered much of a designer, so she's not allowed to access the style sheet. She's smart enough to know, however, that changing this `<h2>` to a hierarchically correct `<h4>` will reduce the visual size of the text, sending the stakeholders into a frenzy.

> *Levels increase with depth. If the DOM ancestry does not accurately represent the level, authors should explicitly define the* `aria-level` *attribute*
> — *WAI-ARIA 1.0*[82]

Using ARIA's `aria-level` property, Anna is able to invisibly correct the underlying structure of the page as described by its heading hierarchy. She simply places the attribute on the `<h2>`, telling screen readers to record and announce the `<h2>` as a fourth-level heading.

```
<h2 aria-level="4">Text that looks big</h2>
```

Although Johnny Paycheck has the kind of talent for visual flair that earns him the title of designer, it is Anna with her knowledge of the specification and appreciation of the medium, who is able to instill structural integrity. Of the two, Anna is the better designer.

Hijacking Links

Imagine you were new to JavaScript and I told you it was possible to add a special line of code which deliberately prevented hyperlinks from doing what hyperlinks were designed for. What would be your first impression? Isn't that just a little reminiscent of the **BIG RED BUTTON** we talked about in chapter 2, which we should never, ever press?

Sometimes there are legitimate reasons for using `return false` or `e.preventDefault` to cancel the normal behavior of hyperlinks in our

82. http://www.w3.org/TR/wai-aria/states_and_properties#aria-level

JavaScript code. For a thumbnail gallery, we could use progressive enhancement and handle activating the link differently depending on whether the user has JavaScript available to them: without JavaScript, a clicked thumbnail link would simply follow the `href` value to the larger picture. With JavaScript available, this default behavior would be suppressed so that we could generate a popup of the larger image in the same page.

The action that the `<a>` element now performs is more akin to a button's action than a link's. To communicate this change of behavior to assistive technology users, we should add the `button` role via the JavaScript which mutates the behavior:

```
<a href="path/to/large/image" role="button">larger picture</a>
```

The benefits of progressive enhancement to accessibility[83] are clear. Since basic access to content is the foundation, a safety net is provided and fewer users are marginalized. It's usually when we start trying to enhance, rather than replace, the default behavior that we start getting in a muddle.

For example, take the employment of a scrolling effect to navigate to anchors within the page. We'll use a nice, reusable piece of jQuery demoed on CSS-Tricks[84]:

```
$(function() {
      $('a[href*=#]:not([href=#])').click(function() {
        if (location.pathname.replace(/^\//,'') ==
        this.pathname.replace(/^\//,'') && location.hostname ==
        this.hostname) {
          var target = $(this.hash);
          target = target.length ? target : $('[name=' +
          this.hash.slice(1) +']');
          if (target.length) {
            $('html,body').animate({
              scrollTop: target.offset().top
            }, 1000);
            return false;
          }
        }
      });
    });
```

83. http://en.wikipedia.org/wiki/Progressive_enhancement#Benefits_for_accessibility
84. http://css-tricks.com/snippets/jquery/smooth-scrolling/

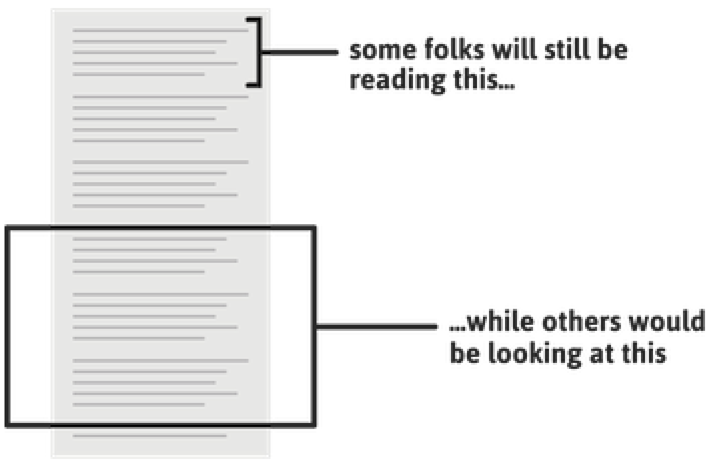
The great thing about this proof of concept is that the jQuery code is based on the correct, standardized markup for creating same-page links. That is, when JavaScript is turned off, the `hash` relationship between the `href` of the link and the `id` of the target can still be relied on to trigger the expected behavior and jump to the target.

```
<a href="#section2">jump to section 2</a>

<!-- lots of content -->

<!-- and lots more content -->

<!-- and even more content -->

<section id="section2">
    <h2>Section 2</h2>
    <!-- etc -->
</section>
```

The only problem is that screen reader users are *not users without JavaScript*. In fact, many of the accessible patterns we will explore later in this book depend on JavaScript to manage attribute values and other characteristics of our widgets.

To consider a screen reader user as not a JavaScript user is an embarrassing misconception — like shouting at someone in a wheelchair because you assume they are deaf.

In the example, the screen reader user's action is undertaken by JavaScript, but all the JavaScript does is animate the scrollbar. It animates the linked anchor into view, but it doesn't explicitly *link* to it in a way most screen readers understand. The unfortunate upshot is that the newly visible section has not been focused. It's on the screen, but that's no good: we haven't technically moved to it.


some folks will still be reading this...
...while others would be looking at this

To make sure the target section is focused and ready to be read and navigated, we need to add two lines to our JavaScript code. The first makes sure the target section can be focused. As a `<section>` element, it can't be focused by default. To do this, we dynamically add a `tabindex` value of `-1`.

```
target.attr('tabindex', '-1');
```

The value of `-1` is perhaps confusing because it sounds like off, or last in array, but it really means the target is *focusable by JavaScript*. A value of `tabindex="0"` would also work, but it would make the element focusable by users, too. Since the section is not interactive, this would depart from expected behavior and confuse some users. Still, anything's better than using positive integers as `tabindex` values, which can quickly result in an illogical tab order as this WebAIM article on `tabindex`[85] attests.

The second line we need to add focuses the target section after it has been scrolled to by the click event:

```
target.focus();
```

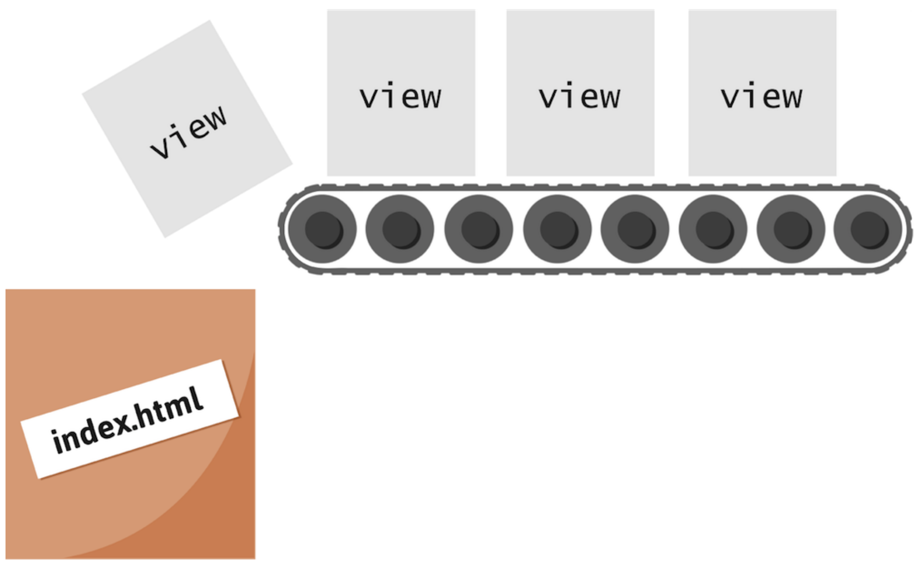
By focusing the section itself, the next thing we can focus on is the first focusable element *inside* the section. Had we not done this, keyboard users would be looking at this section but tabbing through the links of a previous one.

THE PROBLEM WITH VIEWS

Another interface type that suffers from focus-related problems is the JavaScript MV* views interface. Modern JavaScript frameworks like AngularJS base their navigation on views because it enables them to do everything on one page and hold the user data in memory. Views work by rebuilding the HTML in situ whenever the user requests a new screen. It feels very much like going to a new page, but you're really reconstructing the page the user's already on.

> `ngView` *is a directive that complements the* `$route` *service by including the rendered template of the current route into the main layout (*`index.html`*) file.*
> — *AngularJS documentation*[86]

85. http://webaim.org/techniques/keyboard/tabindex
86. http://docs.angularjs.org/api/ngRoute.directive:ngView



This sort of thing is catnip to JavaScript developers, but it doesn't make a whole lot of sense to browsers trying to ensure conventional, accessible navigation. As in the last example, we need to add some more JavaScript to fix the JavaScript. Isn't that *always* the way?

We can write `tabindex="-1"` on the view container manually because that will remain the same. Then, we just need to focus the view container with the `focus()` method whenever the `$route` has changed.

```
<div class="view-container" ng-view tabindex="-1">
    <!-- main page content rebuilt dynamically here -->
</div>
```

Optionally, we could transform the view container into an ARIA live region[87] so screen readers begin speaking the contents of the new view content as soon as they are detected. This would still require the view container to be focused for interaction, though. Another issue is that live regions tend to blurt out all their contents in quick succession unless you are careful.

```
<div class="view-container" ng-view tabindex="-1"
aria-live="assertive">
        <!-- main page content rebuilt dynamically here -->
</div>
```

We shall cover live regions in greater detail in chapter 6 but before that it's time for a little game of *peekaboo*. ❧

87. https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/ARIA_Live_Regions

CHAPTER 5:

Peekaboo

> **peekaboo** *(noun) a game for amusing a baby by repeatedly hiding one's face or body and popping back into view exclaiming 'Peekaboo!'*
> *— Merriam-Webster definition*[88]

Looking at JavaScript-driven web interfaces, by far the most common interaction style is based on showing stuff or hiding it or… oh, that's pretty much it. Strip away the idiosyncratic design, the transition type, scaling, and all the other nuances and — underneath — you're left with one of:

- A thing appearing.
- More than one thing appearing.
- Something(s) appearing and something(s) disappearing *at the same time*.

What about some examples?

- Reveal your site navigation: "Peekaboo!"
- Reveal a dropdown submenu: "Peekaboo!"
- Expand a definition item: "Peekaboo!"
- Switch tabs in a tabbed interface: "Peekaboo!"
- Select accordion menu item: "Peekaboo!"
- Next step in form process: "Peekaboo!"
- Fade in a tooltip: "Peekaboo!"
- Trigger an alert message: "Peekaboo!"
- Open a dialog: "Peekaboo!"

The fact that so many of the so-called rich interactions possible with JavaScript are analogous to an infantile game is no bad thing. First, it stops us from getting big heads about how clever we are as interface

88. http://www.merriam-webster.com/dictionary/peekaboo

designers. More importantly, the common mechanisms uncovered can help us make a finite set of reusable, accessible components.

In magic performance, a hat trick is still a hat trick whether the hat is made of silk or felt. The same can be said of accessible interface design: the content might slide, fade or grow, but the important thing is whether the content — like the white rabbit — is apparent to the audience.

The WAI's ARIA specification recognizes the essential simplicity of typical interface components and provides attributes to define the basic roles, states, properties, and *relationships* of the elements involved. Used together in logical patterns, there's no reason why a connection between the way things can be seen to behave and how they behave should be lost.

In fact, as we shall explore, harnessing ARIA attributes as CSS and jQuery selectors can help us write code which is more reusable and DRY (don't repeat yourself[89]), hopefully saving us effort as well as ensuring accessible solutions.

The Politics Of Hiding

Before showing, comes hiding.

The decision to hide content from some users but not from others is a minefield — an apt metaphor in this case, since much of the danger of explosive mines lies in their being hidden. By definition, hiding from *some* is an act of discrimination. However, there are different kinds of discrimination.

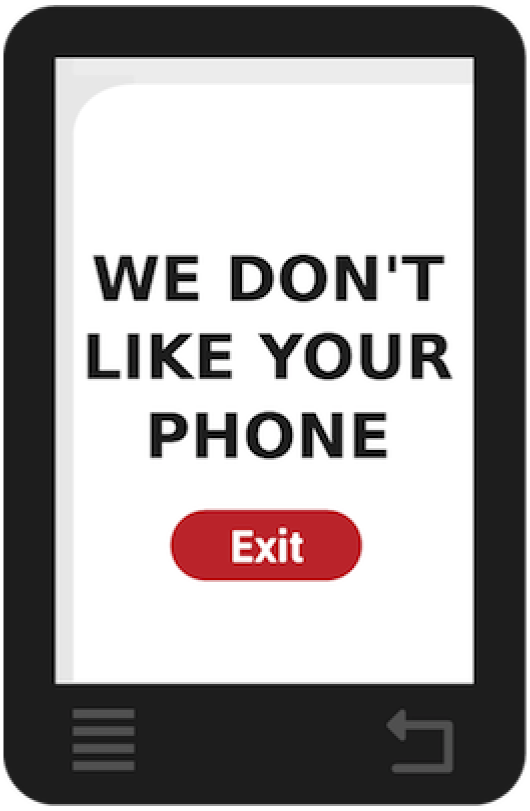
> **discrimination**
> *1. The practice of unfairly treating a person or group of people differently from other people or groups of people*

89. http://en.wikipedia.org/wiki/Don%27t_repeat_yourself

2. The ability to understand that one thing is different from another thing.
— Merriam-Webster definition[90]

BAD DISCRIMINATION

One example of prejudicial treatment (the bad kind of discrimination) would be to make an application work only on certain types of devices. By building a *web* application that only works on Apple's iPhone, you are not creating something in the spirit of the Open Web[91]. Since the free and interoperable[92] technologies you use to make your application are inherently device independent, failing to make its contents available to a wider audience is the worst kind of discrimination: arbitrary discrimination.



Accessibility is not just about addressing specific disabilities, but making sure as many people as possible have access to the same information. There's rarely a good reason to lock people out when openness is a foundational principle of the web.

90. http://www.merriam-webster.com/dictionary/discrimination
91. http://www.w3.org/wiki/Open_Web_Platform
92. http://en.wikipedia.org/wiki/Web_interoperability

GOOD DISCRIMINATION

Other times, discriminating between users is about identifying the types of content they simply can or cannot consume. We have to be sensitive to specific needs and preferences. However, content should *never* be hidden without an alternative version of that content being provided. Under WCAG 2.0's "Perceivable" principle, guideline 1.1 reads "Provide text alternatives for any non-text content so that it can be changed into other forms people need".

THE ARIA-HIDDEN STATE

The `aria-hidden` state is typically used on elements to hide their — and their descendant elements' — content from all users. Hence, the specification for `aria-hidden` recommends tying the visual disappearance of the element(s) to the accessible state:

```
[aria-hidden="true"] { display: none }
```

However, the spec also stipulates that you may, *with caution*, use `aria-hidden` to hide things just from AT users in cases where the content is "redundant or extraneous" to them. Remember: if one form of a piece of content is hidden to some users, another must be revealed to them.

Using aria-hidden Safely

So, when would `aria-hidden` be used to hide content just from screen reader users? Typically, when it eliminates duplication. Roger Johansson wrote about his custom `<select>` elements[93], designed for the purpose of allowing greater design control over items notoriously difficult to style with CSS.

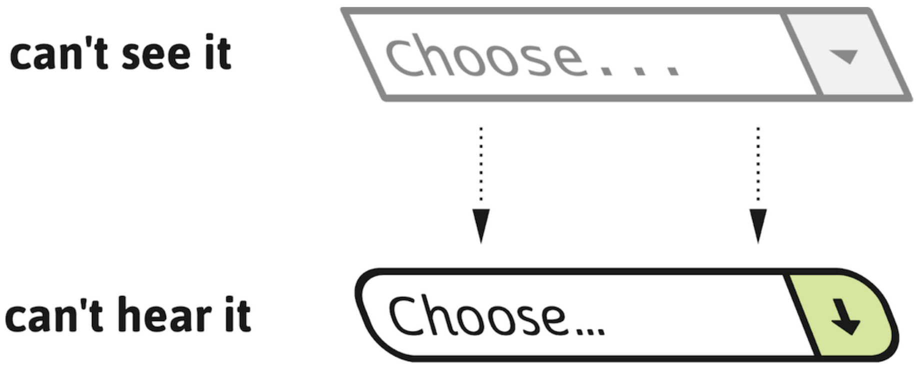
For the technique to work, the `<select>` itself has to be replaced visually with some more malleable `<span>` tags. Because the semantic `<select>` element remains a layer below for accessibility purposes, this results in duplicated content. By hiding the `<span>`-based construction with `aria-hidden`, this duplication is suppressed:

```
<select id="id" class="custom replaced" name="id">...</select>
<span class="custom-select" aria-hidden="true">...</span>
```

The clever part is that the standard `<select>`, hidden with `opacity: 0`, is placed over the top of the custom `<span>`, meaning it captures clicks

93. http://www.456bereastreet.com/archive/201111/an_accessible_keyboard_friendly_custom_select_menu/

from mouse users. That is, though it is visually hidden, it is still interactive to mouse *and* keyboard users. Imitating the full functionality of an element like `<select>` is hard but this method doesn't require you to because you are still interacting with the standard `<select>`, even when you can't see it.



HIDING FROM VIEW

Let's turn the last example on its head and imagine a scenario where we want to provide a text alternative for assistive technology which is invisible on the screen.

Say we have an image containing some represented text. It's not real text so it's not interoperable. It's just the semblance of some text, which many users will read as text. Screen readers cannot deconstruct each pixel that forms that image to understand it and communicate it as text, however. We have to provide some real text for them as an alternative.

The standard way to provide alternative content for images is via the aptly named `alt` attribute. The problem is, when screen readers encounter the image, they will speak "graphic" before the value of the `alt` attribute ("Graphic: My text") or "image" after it. This is fine when the image is a piece of content which should be described as an image but, in our example, the function of the image is textual: reading "graphic" before the text is the equivalent of emblazoning a visible watermark across the image reading "THIS IS AN IMAGE". Nobody needs to know that.

To avoid repeating the same content to either set of users we need to create separate channels: a channel of visual content, and a channel of textual content. Each must be exclusive to its target audience. We need to hide the unwanted channels from the users who don't want them.

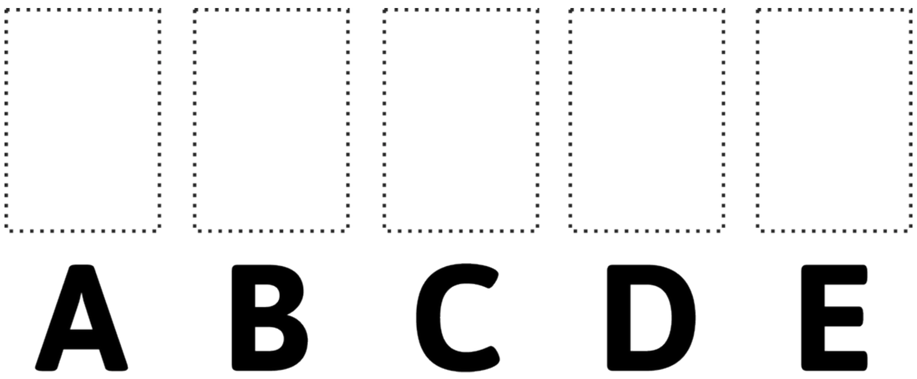
Our first step is to provide an empty or null[94] `alt` attribute: `alt=""`. This makes sure the image is not available to screen readers. Next, we need to include the alternative text and make sure it can't be seen. This is the tricky part.

```
<h2><img src="images/myTitleText.png" alt="" /><span>My title
text</span></h2>
```

Hiding Without Hiding

That leaves us with the problem of what to do if we only want to hide the text visually. There are a number of techniques described by WebAIM[95], mostly involving positioning the text off the screen or using the CSS `clip` property. Another interesting alternative has emerged recently, though, which involves the typography itself.

The Type Team at Adobe has developed a font called Blank[96] which, as the name suggests, is a font of blank letterforms. The cool thing about this is that while the letterforms are invisible, letters are still there, in the HTML code.



Employing Blank is as simple as including a `class` on the `<span>` holding the text with the `font-family` set to the invisible font:

94. http://www.w3.org/TR/WCAG-TECHS/H67.html
95. http://webaim.org/techniques/css/invisiblecontent/
96. http://blog.typekit.com/2013/03/28/introducing-adobe-blank/

```
.visually-hidden {
    font-family: Blank;
}
```

```
<h2><img src="images/myTitleText.png" alt="" /><span
class="visually-hidden">My title text</span></h2>
```

Of course, you will have to load the font via `@font-face` too, which is out of the scope of this book. However, using a version of Adobe Blank[97] which has a reduced character set will improve loading time. Performance is an accessibility issue too!

Naturally, given that we have the ability to embed lovely webfonts, there's really no reason to use an image of text[98] and, in most cases, it should be avoided. Text nodes are highly interoperable, the bread and butter of an accessible interface.

OK, that's enough about hiding already. Let's talk about the big reveal.

Give Me A Clue!

One time you might want to hide content and then show it ("peekaboo!") is when it becomes relevant after a certain action is performed by the user. In this example, we are going to explore how to accessibly reveal hints for entering information into form inputs. There are two advantages to using this kind of *peekaboo* technique:

1. It removes unnecessary clutter from the form which would make it appear complex and confusing.

2. Dynamically revealing hints when interacting with individual form elements draws attention to them and clearly associates them with the chosen input.

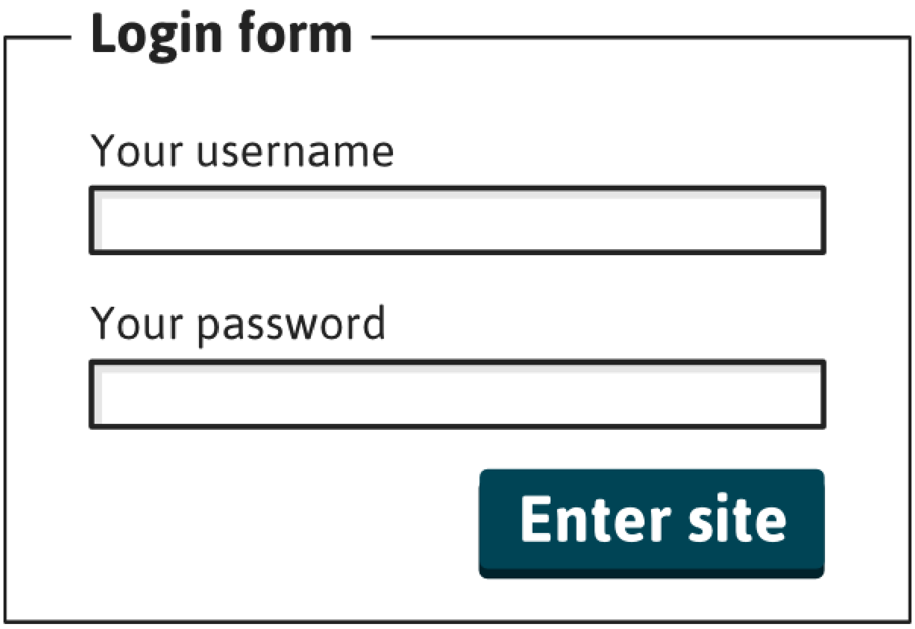
THE SETUP

Let's start by marking up the basic form. Just because we're going to be using a fancy enhancement doesn't mean we can get away with using inaccessible form elements. First things first.

```
<form>
    <fieldset>
```

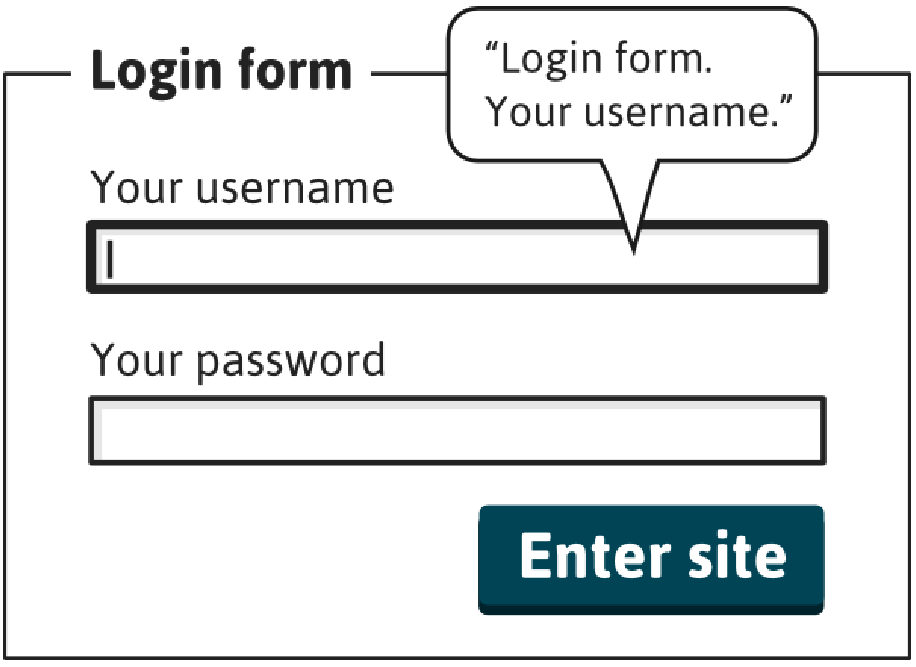
97. https://github.com/stowball/Adobe-Blank
98. http://www.w3.org/TR/UNDERSTANDING-WCAG20/visual-audio-contrast-text-presentation.html

```
        <legend>Login form</legend>
        <div>
            <label for="username">Your username</label>
            <input id="username" type="text">
        </div>
        <div>
            <label for="password">Your password</label>
            <input id="password" type="password">
        </div>
    </fieldset>
    <button type="submit">Enter site</button>
</form>
```



Note the `for` and `id` attributes. These are used to build a relationship between the inputs and their respective labels. You should already be accustomed to coding inputs and labels this way. The reason we do it is so the label is read out when the corresponding input is focused. This way users know which input they're using – always an advantage. Associating an input with a label like this also makes the label interactive, expanding the hit area for the control. This has a similar effect to nesting the input inside the label and helps users target fields more easily.

Many screen readers will prefix the label text with the `<legend>` text, so be careful the two snippets of info make sense when read together ("Login form. Your username").



The final piece in our HTML puzzle is to mark up the hint elements and connect them up with the appropriate inputs. Then we'll code up some CSS to show the hints when needed.

```
<form>
    <fieldset>
        <legend>Login form</legend>
        <div>
            <label for="username">Your username</label>
            <input id="username" type="text"
            aria-describedby="username-hint">
            <div role="tooltip" id="username-hint">&hellip; is
            your email address</div>
        </div>
        <div>
            <label for="password">Your password</label>
            <input id="password" type="password"
            aria-describedby="password-hint">
            <div role="tooltip" id="password-hint" >&hellip; was
            emailed to you</div>
        </div>
    </fieldset>
    <button type="submit">Enter site</button>
</form>
```

Now to break down what the new elements and attributes do.

- `aria-describedby`[99]: Like the label's `for` attribute, this creates a relationship, but this time between the `<input>` and the hint `<div>`. Assistive technologies, via the browser, are told the form field should be described by the hint with this `id` value.

- `role="tooltip"`: Used on the hint itself, this communicates to assistive technologies to treat the hint as a tooltip.

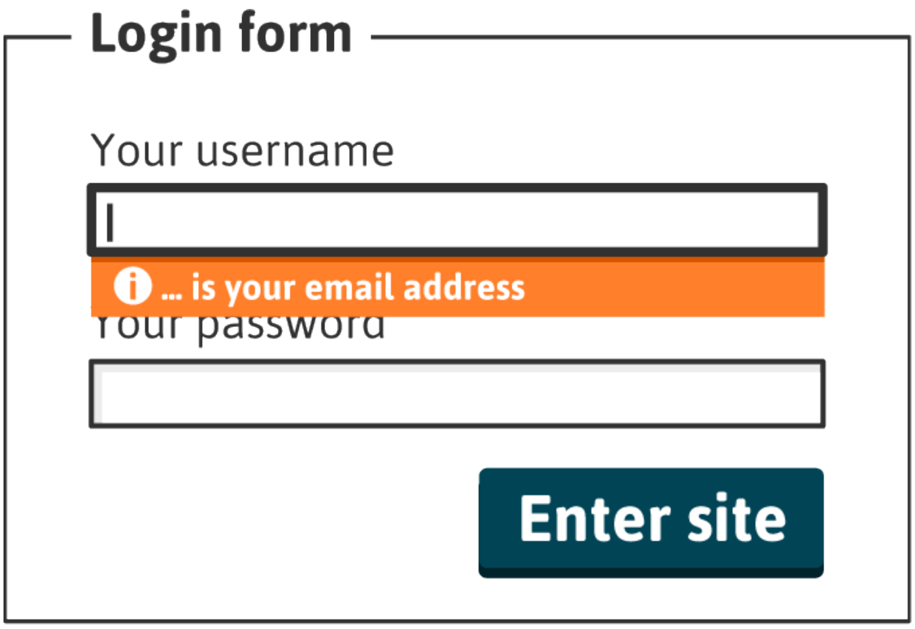
In each case, the hint element is the element after the `<input>` element, meaning we can use a CSS adjacent sibling combinator[100] to show and hide it easily when the input is focused. Note that we are using `display: none` to hide the hint in the first instance because it should be invisible on the screen *and* to screen readers.

```
[role="tooltip"] {
    background: orange;
    color: white;
    padding: 0.25em;
    display: none;
}

input:focus + [role="tooltip"] {
    display: block;
}
```

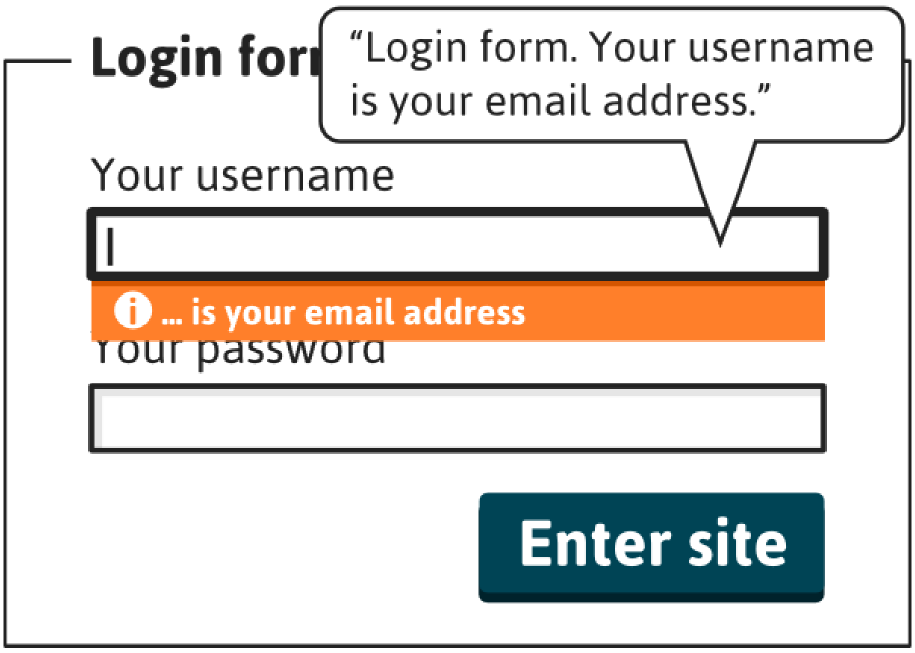
Put together, when you now focus the username `<input>`, first the `<legend>` is announced, followed by the `<input>`'s associated label. After that, the hint text – associated with the `<input>`'s `id` attribute within the `aria-describedby` value – is also spoken. So, you would hear "Login form. Your username (pause) is your email address." All of the information available to a sighted user is disclosed to a screen reader user, read in a logical order.

Note: In many cases, an input with a descriptive label could be considered sufficient and the inclusion of a `<legend>` information overkill. Although many automated testing tools will throw an error if a `<fieldset>` does not contain a `<legend>`, exercise discretion and ask yourself if one is needed for clarity. Because AT will read the legend as well as the label and hint, it's best to keep `<legend>`s short[101].

99. http://www.marcozehe.de/2008/03/23/easy-aria-tip-2-aria-labelledby-and-aria-described-by/
100. http://reference.sitepoint.com/css/adjacentsiblingselector
101. http://www.456bereastreet.com/archive/200904/use_the_fieldset_and_legend_elements_to_group_html_form_controls/



We haven't forgotten keyboard users either! By toggling the visibility of the hint using the `:focus` pseudo class, it will appear when the keyboard user focuses the input, usually by moving to it with the *Tab* key.



What text actually goes into any hint element is a question of context and the state of the application's data model. This is a job for JavaScript or a server-side programming language to determine. In any case, with just HTML, CSS, and some ARIA, we've established an accessible, reusable pattern for communicating those hints.

A working demo[102] similar to this example is available.

102. http://heydonworks.com/practical_aria_examples/#input-tooltip

Progressively Collapsible

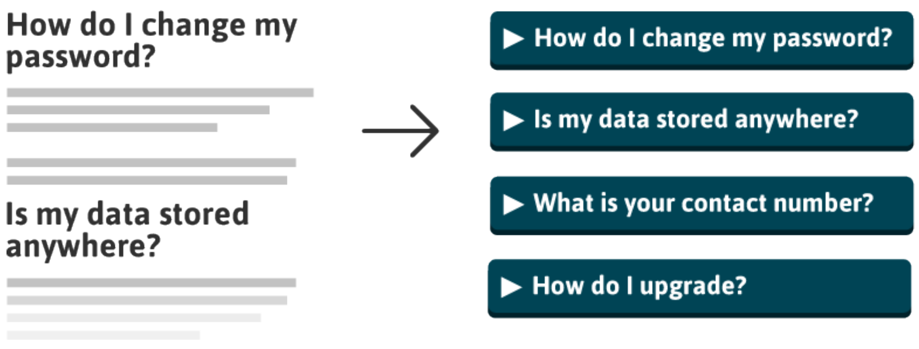
It's time we looked at a more complex example of the peekaboo game using some progressive enhancement.

Let's imagine we've created a page of frequently asked questions (FAQs). People ask us a lot of questions about our web application, so this page is quite long. Each question is marked up with an `<h2>` and the answer in a number of paragraphs, images and other content following that `<h2>`.

```
<h2>How do I change my password?</h2>
<p>Lorem etc etc etc.</p>
<p>etc.</p>

<h2>Is my data stored anywhere?</h2>
<p>Lorem etc etc etc.</p>
<img src="images/diagram.png" alt="data model" />
<p>etc.</p>

<h2>What is your contact number?<h2>
<p>Lorem etc etc etc.</p>
<p>Some text with a <a href="http://www.example.com/article">
link</a> in it etc.</p>
```

This markup is in pretty good shape, semantically speaking. We've used proper headings to mark up the questions, helping to create a structural hierarchy that is both visually apparent and ensures screen reader users can jump from question to question with ease, via the 2 quick key.

The drawback is the length of the page, making it difficult to break down and get an overview. Since `<h2>` elements are not focusable via the keyboard, there's also no way for keyboard users to easily navigate between independent questions. It's a solid foundation, but we'd like to progressively enhance the experience for the many people who have access to JavaScript.


How do I change my password?
Is my data stored anywhere?
▶ How do I change my password?
▶ Is my data stored anywhere?
▶ What is your contact number?
▶ How do I upgrade?

THE HTML

What we'd like to do is collapse the content of each section so that just the questions remain. Then we'd like to enable users to reveal individual answers based on a click, tap, or keystroke. Overall, identifying questions and having them answered would be a more pleasant experience.

Many of the helper elements and attributes we are going to use should be added by jQuery because they are irrelevant in pages where JavaScript is not available. It also means we have less markup to write manually, which is good. Let's look at the HTML of one question and answer panel which has been adapted by our script.

```
<h2><button aria-expanded="false"
aria-controls="how-do-I-change-my-password">How do I change my
password?</button></h2>
<div id="how-do-I-change-my-password" aria-hidden="true">
    <p>Lorem ipsum with a <a href="http://example.com">link
    thrown in</a> etc.</p>
    <p>etc.</p>
</div>
```

1. We have wrapped all of the answer's content in a common `<div>` so we only have one thing to expand or collapse. To do this in jQuery is simple: `$('h2').nextUntil('h2').wrapAll('div')`

2. We have wrapped the content of the `<h2>` with a button which will be used to perform the expansion of the answer `<div>`. We could use a `role` of `button` on the `<h2>` but we'd lose our heading semantics (remember the second rule of ARIA use[103]).

3. We have established a relationship between the heading button and the hidden answer using the `aria-controls` property[104] and a generated `id` so assistive technology knows which element is affected by which. To generate a valid slug from the question text like I have, you could use this tiny helper plugin[105] or generate a random key. To establish the relationship, it doesn't matter what the `id` is, so long as it's unique.

4. The button includes the `aria-expanded` state[106] to indicate whether or not the corresponding answer is available (displayed and able to be read). To start with, it is set to `false`, naturally.

---

103. http://www.w3.org/TR/aria-in-html/#second-rule-of-aria-use
104. http://www.w3.org/TR/wai-aria/states_and_properties#aria-controls
105. https://github.com/pmcelhaney/jQuery-Slugify-Plugin
106. http://www.w3.org/TR/wai-aria/states_and_properties#aria-controls

5. We have told assistive technologies that the unexpanded answer `<div>` should be hidden with `aria-hidden`. With `aria-hidden`, we are trying to ensure the answer is not stumbled on unexpectedly.

THE CSS

Just a few notes on the CSS. As described in chapter 2, "It's All About Buttons", it's good form to tie the presentation to the accessible states.

> *Tie CSS `display` property to WAI-ARIA hidden state. This is important for assistive technologies who communicate directly with the user agent's DOM versus a platform accessibility API supported by the user agent.*
> — *WAI-ARIA Authoring Practices*[107]

By doing this we can dispense with extraneous `class` attributes and values, but also make sure that visual changes reflect real state changes. For example, we collapse and expand each answer by toggling the `aria-hidden` state:

```css
[aria-hidden] {
    display: none;
}

[aria-hidden="false"] {
    display: block;
}
```

Additionally, we use some `:before` pseudo content to place an arrow next to the question text. The direction this arrow points — right for collapsed and down for expanded — is tied to the value of the `aria-expanded` state (false or true).

```css
[aria-expanded]:before {
    content: '\25ba\0020';
}

[aria-expanded="true"]:before {
    content: '\25bc\0020';
}
```

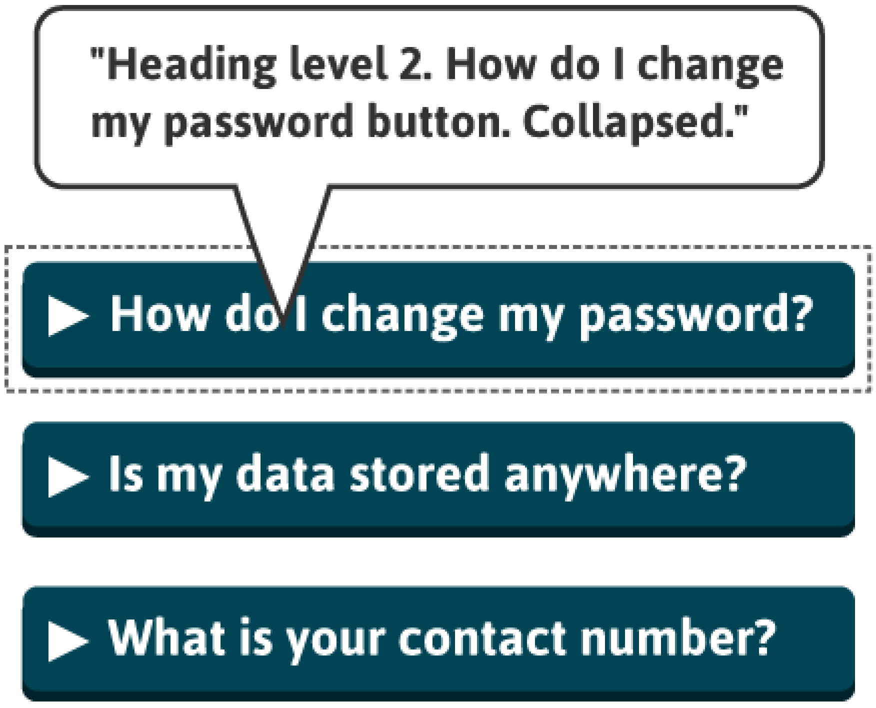
Using Unicode symbols for the arrows is much less complex and more efficient than inserting graphics or background images. It also means

107. http://www.w3.org/WAI/PF/aria-practices/#docmgt

there's no loss in quality should a user decide to zoom the page. Tip: the `\0020` encoding corresponds to SPACE U+0020[108] and should respect the word spacing of your heading typography. Note that some screen readers will announce "*right pointing arrow*" or similar.

WHAT HAPPENS THEN?

Well, once the markup has been set up, all the JavaScript does is handle button clicks to toggle the `aria-expanded` and `aria-hidden` states on the button and the corresponding panel respectively. What we actually get from adding and changing all these attributes is a lot more interesting.


"Heading level 2. How do I change my password button. Collapsed."
▶ How do I change my password?
▶ Is my data stored anywhere?
▶ What is your contact number?

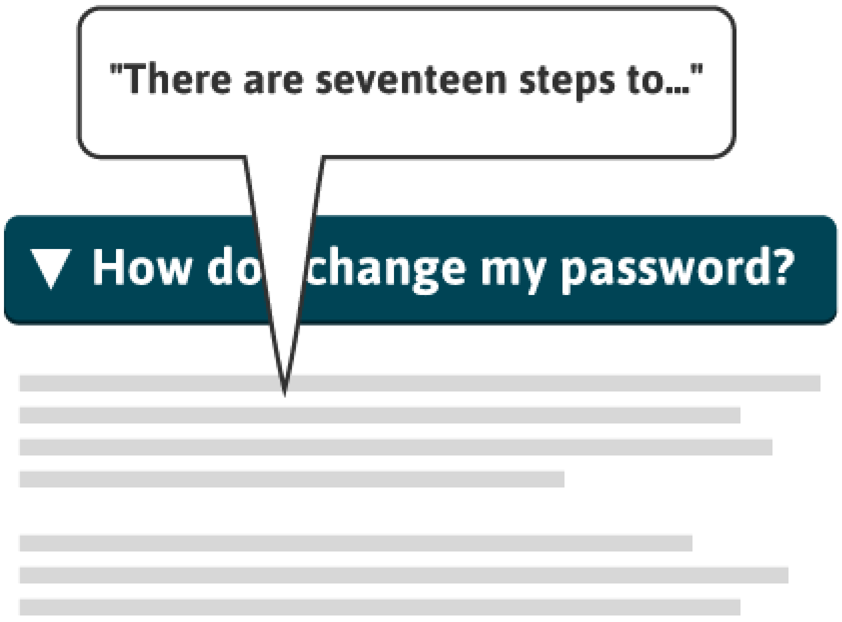
When a button representing a question is focused, screen readers announce four important pieces of information:

- The heading (question) text itself.
- The level of that heading (2, in this example).
- That we are focused on a button.
- Whether this button represents a collapsed or expanded state.

When we press the `<button>`, its `aria-expanded` state it toggled to `false` and the `aria-hidden` state of the corresponding panel is also

108. http://www.fileformat.info/info/unicode/char/0020/index.htm

`false`. The announcement, "*How do I change my password button expanded*", tells the user that the content is now available to read. Now, they can move to that content, typically by using the down arrow key.


"There are seventeen steps to..."
▼ How do I change my password?

JAWS provides one additional piece of information: "*Use the JAWS key plus ALT plus M to move to the controlled element.*" JAWS has made explicit the relationship formed using `aria-controls` and has offered the user a way to navigate to the expanded content. Because the expanded content appears directly after the controller in our example, this isn't really needed. It would be helpful, however, in situations where the expanded content is elsewhere in the page.

A working demo[109] of this example is available.

ONE PATTERN TO RULE THEM ALL

In the last example, we took some semantic HTML, then used some ARIA and a pinch of JavaScript to progressively enhance it, turning a page of content into a quick reference. The accessibility of this technique is twofold:

1. It makes the content accessible to users whose JavaScript has an error, hasn't loaded or is blocked by security.
2. It makes the JavaScript-enhanced view of the same content usable by keyboard navigators and screen reader users.

109. http://heydonworks.com/practical_aria_examples/#progressive-collapsibles

At the heart of the enhanced view there is very little in terms of systemic complexity and its applications are highly varied. The same ARIA expand/collapse metaphor could be applied to accordion menus (where only one region is expanded at a time), or to reveal navigation menus hidden behind the “☰” icon or *navicon*.

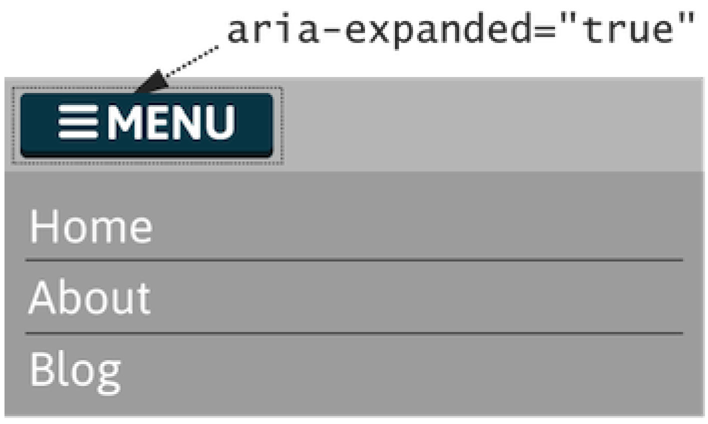
Navicons

The only significant difference between the previous example and the next one for expandable navigation menus (via the navicon) would be the way it degrades to basic HTML. Without JavaScript, we should see a simple same-page link to the navigation landmark:

```
<a href="#navigation">Menu</a>
<!-- some other markup, possibly -->
<nav id="navigation" role="navigation">
    <ul>
            <li><a href="/">home</a></li>
            <li><a href="/about">about</a></li>
            <li><a href="/blog">blog</a></li>
    </ul>
</nav>
```

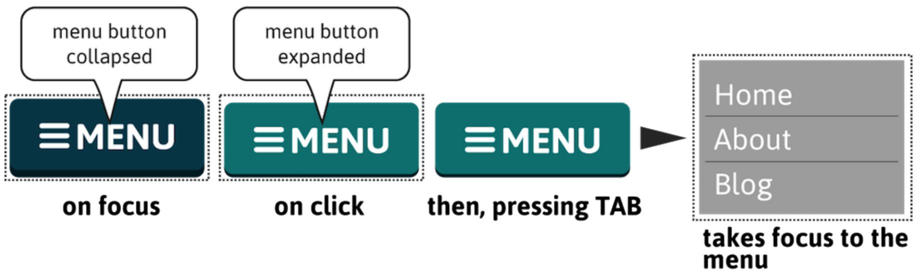
With JavaScript available, we would disguise the link as a `button` using ARIA’s `role="button"`. This makes sense because we would be changing the action from going to the navigation to opening it. That is, our link would be made to `return false`. The rest should be familiar to you already…

```
<a href="#navigation" role="button" aria-controls="navigation"
aria-expanded="false">Menu</a>
<!-- some other markup, possibly -->
<nav id="navigation" role="navigation" aria-hidden="true">
    <ul>
                <li><a href="/">home</a></li>
                <li><a href="/about">about</a></li>
                <li><a href="/blog">blog</a></li>
        </ul>
</nav>
```



Notes

- When the menu button is pressed, the screen reader should confirm that it is now a "Menu button expanded". This lets screen reader users know they can now use the navigation menu.
- We would attach a `keypress` event to `$([aria-expanded])` so that when the user presses *Tab*, they would focus the first item in the navigation menu. We have to do this programmatically because the menu is not necessarily next in the tab order (other interactive elements might exist between the menu button and the menu, depending on how your DOM is organized).
- Note that using `aria-hidden` to hide the navigation landmark will make it inaccessible to screen reader users via the landmarks dialog. In this off-canvas navigation demo[110], the problem is solved by cloning the landmark and hiding it visually at the foot of the page.



TEST.CSS

By removing classes from our design, this off-canvas menu pattern is reduced to its essential parts. It should be difficult to code badly without knowing because all of the ARIA attributes would also be used as

110. http://heydonworks.com/practical_aria_examples/#hamburger

styling hooks. Nonetheless, it's better to be safe than sorry. Add these rules to your *test.css* file:

```
#navigation:not([role="navigation"]):after {
    background: red;
    color: white;
    content: 'Warning: You appear to be linking to a principal
    navigation block. Make sure it has the navigation ARIA role';
}

[aria-controls="navigation"]:not([href="#navigation"]):after {
    background: red;
    color: white;
    content: 'Warning: When JavaScript is turned off, this should
    be a link and should go to the navigation landmark';
}
```

Note: When just displaying the navicon symbol, without "menu" appended to it, you should include an `aria-label` attribute with a value of something like "navigation menu" as discussed in chapter 2, "It's All About Buttons". Though this would make the button recognizable to screen reader users, it does not make up for the fact that the symbol may be ambiguous to many others. As Luis Abreu attests in "Why And How To Avoid Hamburger Menus"[111], this side drawer pattern can be damaging to usability across the board. If a design pattern is demonstrably unusable, making it more accessible doesn't help: all you achieve is better access to something nobody wants!

Can I Get A Tab?

You won't get far on the web these days without stumbling on some sort of tabbed interface. You know the kind of thing: a line of tabs, like those used in a filing cabinet, with each corresponding to their own pane or panel of content. It's a popular pattern because it allows users to browse and switch between content, excluding from view anything they're not interested in.

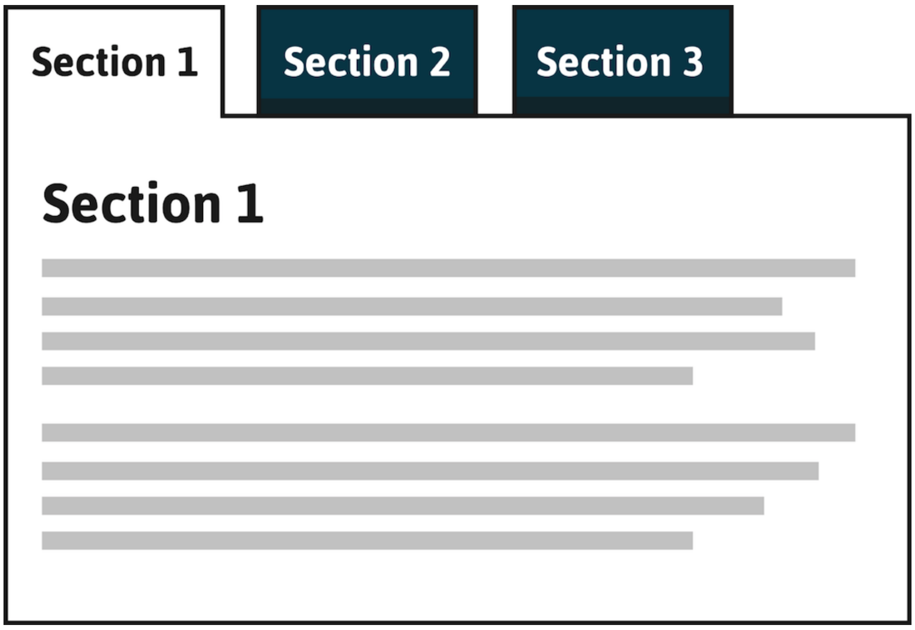
In fact, tabbed interfaces are so popular, it's tempting to think of them as *done*: the JavaScript to show and hide panels is easy to write and easier to steal, which just leaves the visual design to be pondered.

- Should the tabs be rounded?

111. http://lmjabreu.com/post/why-and-how-to-avoid-hamburger-menus/

- Maybe some overlap would look nice.
- How do I make the selected one look like it's on top?
- Is pink really the best color for this?

Important questions I'm sure, but a good-looking tabbed interface does not a good tabbed interface make – not on its own. Is the underlying structure properly semantic and accessible? The JavaScript shows one panel and hides the others, but is this action really communicated to everyone? Can everyone perform that action in the first place?



Recognizing the prevalence of the tabbed interface and conceding its relative complexity, ARIA provides a number of dedicated roles and associated properties and states for building composite tab widgets to make this kind of navigation a reality to screen reader users. By combining these attributes with some carefully designed keyboard support, we can finally fulfill the real brief of making an accessible tabbed interface.

We only have to get this right once. After that, you can reuse the pattern and restyle it as much and as many times as you like.

THE SETUP

In Léonie Watson's presentation "A Rock 'n' Roll Guide To HTML5 And ARIA[112]," the section on tabbed widgets is prefaced with You Ain't Seen Nothin' Yet by Bachman Turner Overdrive. It certainly is more ambi-

112. http://www.slideshare.net/LeonieWatson/generate-2013-09

tious than anything else we've tried so far, but it's not so bad when you break it down.

We're going to employ some progressive enhancement again, so let's start with the basic HTML. All tabbed interfaces should begin life as a list of navigation links that take you to different parts of some content. Remember the animated same-page links we fixed in chapter 4? That's the sort of markup we're after. Perfectly serviceable.

```
<ul>
   <li><a href="#section1">Section 1</a></li>
   <li><a href="#section2">Section 2</a></li>
   <li><a href="#section3">Section 3</a></li>
</ul>
<section id="section1">...</section>
<section id="section2">...</section>
<section id="section3">...</section>
```

(**Note:** Tabbed interfaces pertain to interfaces that allow the user to switch between content sections on the same page. Though you may style your main website navigation to look like a set of tabs, this does *not* count as a true tabbed interface, semantically speaking.)

That's a good start, but it's not a tabbed interface. Not even if we style it to look like one. Let's incorporate the ARIA attributes and extend the semantics to create our accessible widget.

```
<ul role="tablist">
   <li role="presentation"><a href="#section1" role="tab"
   aria-controls="panel1" aria-selected="true">Section 1</a></li>
   <li role="presentation"><a href="#section2" role="tab"
aria-controls="panel2">Section 2</a></li>
   <li role="presentation"><a href="#section3" role="tab"
   aria-controls="panel2">Section 3</a></li>
</ul>
<section id="section1" role="tabpanel">...</section>
<section id="section2" role="tabpanel"aria-hidden="true">
...</section>
<section id="section3" role="tabpanel" aria-hidden="true">
...</section>
```

Yikes! – that's a lot of new attributes. Here's how they work together:

- **tablist** (role): This is a "composite[113]", meaning it groups navigational items together as part of a widget. It houses our tabs.
- **tab** (role): One tab helping to make up our tablist.
- **aria-selected** (state): Indicates the selected or open tab in a way that's infinitely more accessible than `class="selected"` (which communicates nothing to assistive technology).
- **tabpanel** (role): Defines one tab's associated panel of content. An accessible relationship is created between each `tab` and its `tabpanel` via `aria-controls`, as in previous examples.
- **presentation** (role): A special role which removes the meaning of the element. Practically speaking, it turns `<li>` into `<>`. Without removing them altogether, we can stop the `<li>`s being identified when the roles and states are added and we don't need them anymore.

MANAGING FOCUS

Sticking a few fancy new attributes on some elements is easy enough, but thinking about managing focus[114] and how different users interact with the widget is more of a stretch. Fortunately, keyboard control boils down to just two things here. We want to:

1. Switch between (that is, change focus to) adjacent tabs using the left and right arrow keys — an action which will also reveal the corresponding tab panel.
2. Switch between the active (`aria-selected`) tab and its contolled `tabpanel` using the *Tab* key, and back again with *Shift* + *Tab*.

For the second item, we could just make the `tabpanel` itself focusable using `tabindex="0"` but focusing the panel itself stops some user agents from announcing "tab panel" for context. Instead, we'll move focus to the first element inside the panel. In this example, that will always be a heading because `section`s should have headings. In which case, "Heading level 3, [Heading text], tab panel" should be announced on focus.

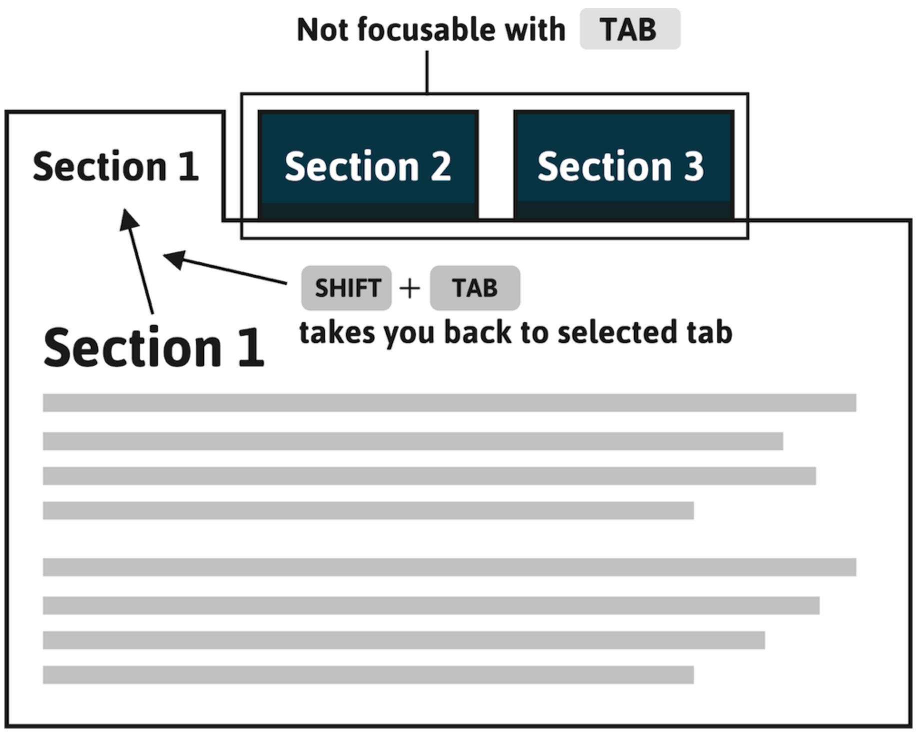
The panels that are hidden (using `display: none` and `aria-hidden="true"`) are removed from the tab order, leaving just the open one which we can jump into. That is, the *visible* `<h3>` (the first one below) with `tabindex="0"` is next in the tab order.

113. http://www.w3.org/TR/wai-aria/roles#composite
114. http://www.w3.org/TR/wai-aria/usage#managingfocus

```
<section id="section1" role="tabpanel">
    <h3 tabindex="0">Section 1</h3>
</section>
<section id="section2" role="tabpanel" style="display: none"
aria-hidden="true">
    <h3 tabindex="0">Section 2</h3>
</section>
<section id="section3" role="tabpanel" style="display: none"
aria-hidden="true">
    <h3 tabindex="0">Section 3</h3>
</section>
```

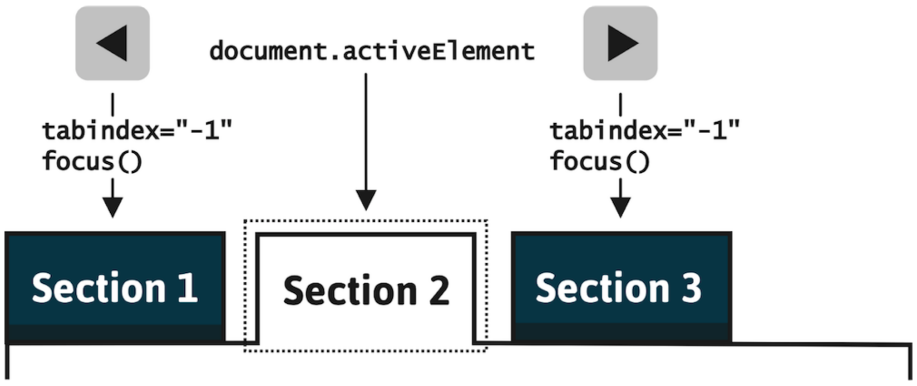
Moving between the open `tabpanel` and the `tab` which names it isn't possible unless the tabs between the panel and this tab have ceased to be focusable via the *Tab* key. We make this happen by giving all but the `aria-selected` tab a `tabindex` of `-1`, removing them from the tab order.

```
<ul role="tablist">
   <li role="presentation"><a href="#section1" role="tab"
   aria-controls="panel1" aria-selected="true">Section 1</a></li>
   <li role="presentation"><a href="#section2" role="tab"
   aria-controls="panel2" tabindex="-1">Section 2</a></li>
   <li role="presentation"><a href="#section3" role="tab"
   aria-controls="panel2" tabindex="-1">Section 3</a></li>
</ul>
```



We still want the other tabs to be focusable (it's not much of a tabbed interface otherwise!) but we want to switch between them with the arrow keys only. Using JavaScript, we bind functions to the keys which programmatically focus either the previous or next tab. In pseudocode it would be something like this:

```
if (a key was pressed while a tab was focused) {
    if (the right arrow was pressed and a right tab exists) {
        focus the next right tab;
        make the next right tab's tabindex '0' and this one's
        '-1';
    }
    if (the left arrow was pressed and a left tab exists) {
        focus the next left tab;
        make the next left tab's tabindex '0' and this one's '-1';
    }
}
```



A working demo[115] of this tabbed interface example is available. It is also worth reading Jason Kiss's research[116] into ARIA tabbed interfaces for more detail and some alternative solutions.

TEST.CSS

A few rules to make sure our interface's HTML is on the right track. You may want to add more of your own rules here, especially if your development team insist on using classes for the styling and state switching! You could test for the presence of the correct ARIA attributes with selectors like `.tab:not([role="tab"]):after {}`.

115. http://heydonworks.com/practical_aria_examples/#tab-interface
116. http://accessibleculture.org/articles/2010/08/aria-tabs/

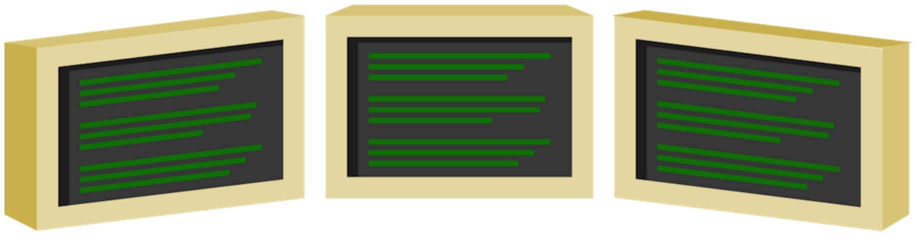
```
[role="tablist"] a:not([role="tab"]):after {
    background: red;
    color: white;
    content: 'Warning: All links inside a tablist should be
    defined as tabs, using the tab ARIA role';
}

[role="tabpanel"]:not([id]):after {
    background: red;
    color: white;
    content: 'Warning: Each tabpanel should be identified with an
    id attribute';
}

[role="tab"]:not([aria-controls]):after {
    background: red;
    color: white;
    content: 'Warning: Each tab should explicitly control
    a tabpanel using the aria-controls attribute';
}
```

OK, now we're done with showing and hiding things. Well, we've got a general grasp of doing it accessibly anyway. Next up: live regions and how they can help us keep users informed about changes as they happen within the application. Changes that have been made either by the user or by the application on their behalf. We'll also build a modal dialog, which intercepts an action the user is trying to make and gives them the option to back out. It's all about keeping open the communication channel shared by the application and the user. ❧

CHAPTER 6:

It's Alive!

Picture the scene: it's a day like any other and you're at your desk, enclosed in a semicircular bank of monitors that make up your extended desktop, intently cranking out enterprise-level CSS for MegaDigiSpace-Hub Ltd. You are one of many talented front-end developers who share this floor in your plush London office.



You don't know it, but a fire has broken out on the floor below you due to a "mobile strategist" spontaneously combusting. Since no expense was spared on furnishing the office with adorable postmodern ornaments, no budget remained for installing a fire alarm system. It is up to the floor manager in question to travel throughout the office, warning individual departments in person.

He does this by walking silently into each room, holding a business card aloft with the word "fire" written on it in 12pt Arial for a total of three seconds, then leaving. You and the other developers — ensconced behind your monitors — have no idea he even visited the room.

This book is, for the most part, about making using your websites and applications accessible. That is, we're concerned with everyone being able to do things with them easily. However, it is important to acknowledge that when something is done (or simply happens), something else will probably happen as a result: there are actions and reactions.

> *When one body exerts a force on a second body, the second body simultaneously exerts a force equal in magnitude and opposite in direction to that of the first body.*
> *— Newton's third law of motion (Newton's laws of motion, Wikipedia[117])*

117. http://en.wikipedia.org/wiki/Newton%27s_laws_of_motion

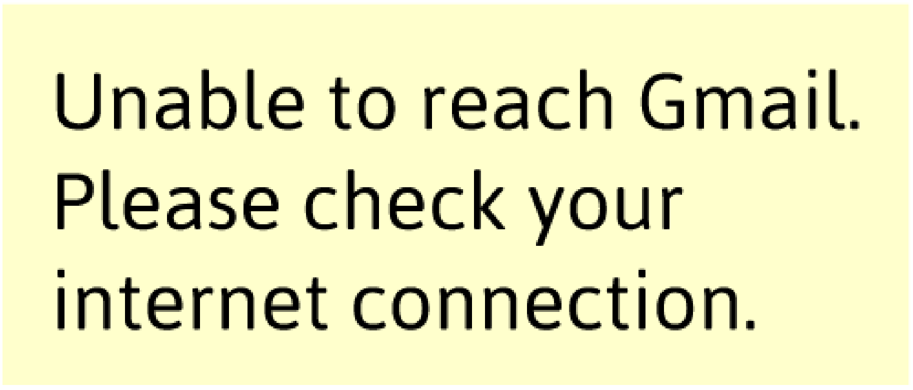
Providing feedback to users, to confirm the course they've taken, address the result of a calculation they've made or to insert helpful commentary of all sorts, is an important part of application design. The problem which needs to be addressed is that interrupting a user visually, by making a message appear on screen, is a silent occurrence. It is also one which — in the case of dialogs — often involves the activation of an element that originates from a completely remote part of the document, many DOM nodes away from the user's location of focus.

To address these issues and to ensure users (unlike the poor developers in the introductory story) get the message, ARIA provides live regions[118]. As their name suggests, live regions are elements whose contents may change in the course of the application's use. They are living things, so don't always stand still. By adorning them with the appropriate ARIA attributes, these regions will interrupt the user to announce their changes as they happen.

We shall look in this chapter at various uses for different live region `role`s and other, associated attributes. Mostly this involves feedback on actions taken by the user. As in the following example, we will also look at how to alert users to changes which they *didn't* ask for, but — like the building being on fire — really ought to know about anyway.

Alert!

Perhaps the only thing worse than a fire that could happen to the office of a web development company would be losing connectivity to the web. Certainly, if I was working using an online application, I'd like to know the application will no longer behave in the way I expect and perhaps store my data properly. This is why Google Mail inserts a warning whenever you go offline. As noted in Marco Zehe's 2008 blog post[119], Google was an early adopter of ARIA live regions.


Unable to reach Gmail. Please check your internet connection.

118. https://developer.mozilla.org/en-US/docs/Accessibility/ARIA/ARIA_Live_Regions
119. http://www.marcozehe.de/2008/08/04/aria-in-gmail-1-alerts/

We are going to create a script which tests whether the user is online or off and uses ARIA to warn screen reader users of the change in this status so they know whether it's worth staying at their desk or giving up and going for a beer.

THE SETUP

For live regions, ARIA provides a number of values for both the `role` and `aria-live` attributes. This can be confusing because there is some crossover between the two and some screen readers only support either the `role` or `aria-live` alternatives. It's OK, there are ways around this.

At the most basic level, there are two common types of message:

1. "This is pretty important but I'm going to wait and tell you when you're done doing whatever it is you're doing."
2. "Drop everything! You need to know this now or we're all in big trouble. AAAAAAAAAAGHH!"

Mapped to the respective `role` and `aria-live` attributes, these common types are written as follows:

1. "This is pretty important but I'm going to wait and tell you when you're done doing whatever it is you're doing." (`aria-live="polite"` or `role="status"`)
2. "Drop everything! You need to know this now or we're all in big trouble. AAAAAAAAAAGHH." (`aria-live="assertive"` or `role="alert"`)

When marking up our own live region, we're going to maximize compatibility by putting both of the equivalent attributes and values in place. This is because, unfortunately, some user agents do not support one or other of the equivalent attributes. More detailed information on maximizing compatibility[120] of live regions is available from Mozilla.

Since losing internet connectivity is a major disaster, we're going to use the more aggressive form.

```
<div id="message" role="alert" aria-live="assertive"
class="online">
    <p>You are online.</p>
</div>
```

120. https://developer.mozilla.org/en-US/docs/Accessibility/ARIA/ARIA_Live_Regions

The code above doesn't alert in any way by itself—the contents of the live region would have to dynamically change for that to take place. The script below will run a check to see if it can load *test_resource.html* every three seconds. If it fails to load it, or it has failed to load it but has subsequently succeeded, it will update the live region's `class` value and change the wording of the paragraph. If you go offline unexpectedly, it will display `<p>There's no internets. Time to go to the pub!</p>`.

The change will cause the contents of that `#message` live region to be announced, abruptly interrupting whatever else is currently being read on the page.

```
// Function to run when going offline

var offline = function() {
  if (!$('#message').hasClass('offline')) {
    $('#message') // the element with [role="alert"] and
    [aria-live="assertive"]
      .attr('class', 'offline')
      .text('There\'s no internets. Go to the pub!');
  }
}

// Function to run when back online

var online = function() {
  if (!$('#message').hasClass('online')) {
    $('#message') // the element with [role="alert"] and
    [aria-live="assertive"]
      .attr('class', 'online')
      .text('You are online.');
  }
}

// Test by trying to poll a file

function testConnection(url) {
    var xmlhttp = new XMLHttpRequest();
    xmlhttp.onload = function() {
      online();
    }
    xmlhttp.onerror = function() {
      offline();
    }
    xmlhttp.open("GET",url,true);
    xmlhttp.send();
}

// Loop the test every three seconds for "test_resource.html"

function start() {
  rand = Math.floor(Math.random()*90000) + 10000;
  testConnection('test_resource.html?fresh=' + rand);
  setTimeout(start, 3000);
}

// Start the first test

start();
```

"Alert: There's no internets.
Go to the pub!"
There's no internets.
Go to the pub!

There are more comprehensive ways to test to see if your application is online or not, including a dedicated script called offline.js[121], but this little one is included for context. Note that some screen readers will prefix the announcement with "Alert!", so you probably don't want to include "Alert!" in the actual text as well, unless it's really, *really* important information.

There is a demo of this example[122] available.

121. http://github.hubspot.com/offline/docs/welcome/
122. http://heydonworks.com/practical_aria_examples/#offline-alert

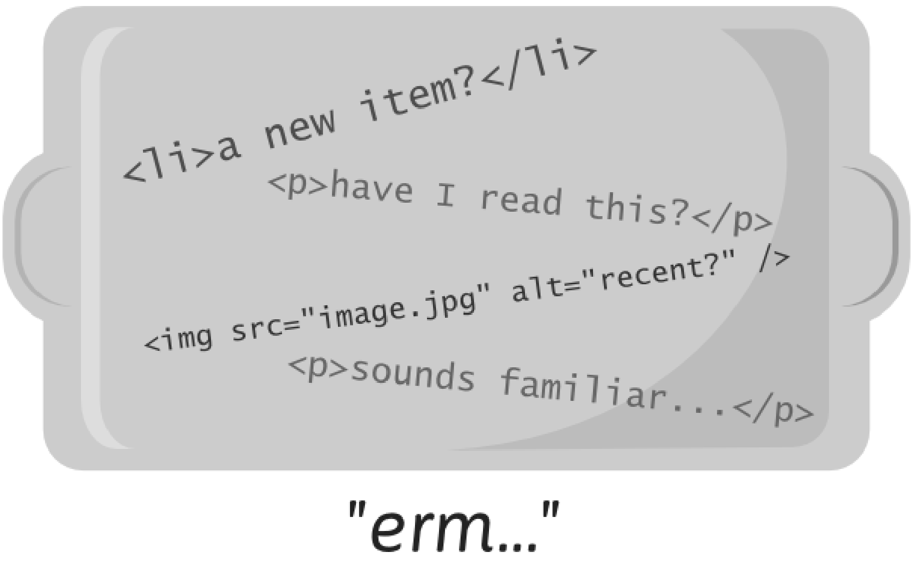
TEST.CSS

We would like to maximize compatibility of live regions across browsers and assistive technologies. We can add a rule in our *test.css* to make sure equivalent attributes are all present like so:

```
[role="status"]:not([aria-live="polite"]),
[role="alert"]:not([aria-live="assertive"]) {
        content: 'Warning: For better support, you should include
        a politeness setting for your live region role using the
        aria-live attribute';
}

[aria-live="polite"]:not([role="status"]),
[aria-live="assertive"]:not([role="alert"]) {
        content: 'Warning: For better support, you should
        include a corresponding role for your aria-live
        politeness setting';
}
```

I Want The Whole Story

Taken out of context, I must seem so strange.
— Fire Door by Ani DiFranco

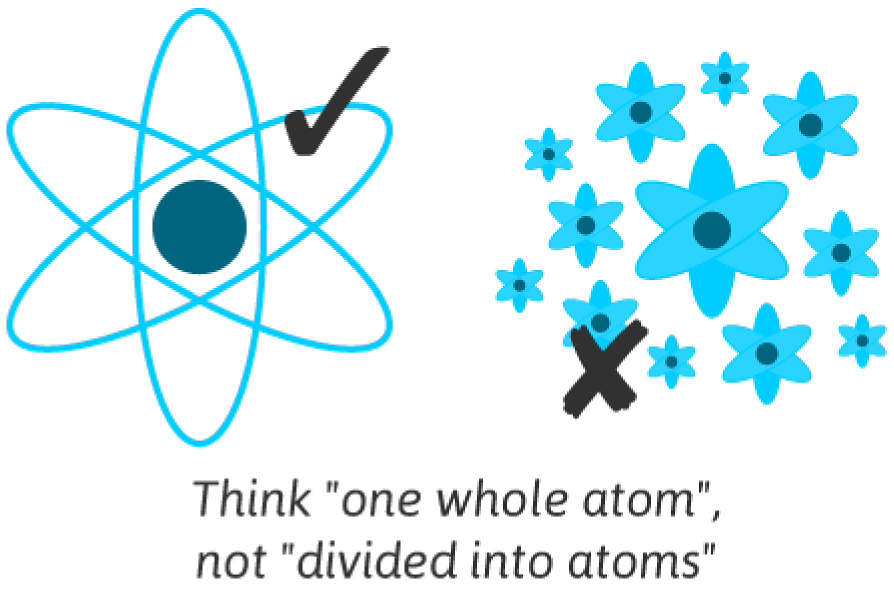


<li>a new item?</li>
<p>have I read this?</p>
<img src="image.jpg" alt="recent?" />
<p>sounds familiar...</p>
"erm..."

By default, when the contents of a live region alter, only the nodes (HTML elements, to you and me) which have actually changed are announced. This is helpful behavior in most situations because you don't want a huge amount of content reread to you just because a tiny part of it is different. In fact, if it's all read out at once, how would you tell which part had changed? It would be like the memory tray game where you have to memorize the contents of a tray to recall which things were removed.

In some cases, however, a bit of context is desirable for clarification. This is where the `aria-atomic` attribute comes in. With no `aria-atomic` set, or with an `aria-atomic` value of `false`, only the elements which have actually changed will be notified to the user. When `aria-atomic` is set to `true`, all of the contents of the element with `aria-atomic` set on it will be read.

The term *atomic* is a little confusing. To be `true` means to treat the contents of this element as one, indivisible thing (an atom), not to smash the element into little pieces (atoms). Whether or not you think atomic is a good piece of terminology, the expected behavior is what counts and it is the first of the two behaviors which is defined.



Gez Lemon offers a great example of `aria-atomic`[123]. In his example, we imagine an embedded music player which tells users what the currently playing track is, whenever it changes.

```
<div aria-live="polite" role="status" aria-atomic="true">
  <h3>Currently playing:</h3>
  <p>Jake Bugg - Lightning Bolt</p>
</div>
```

Even though only the name of the artist and song within the paragraph will change, because `aria-atomic` is set to `true` the whole region will

123. http://juicystudio.com/article/wai-aria_live-regions_updated.php

be read out each time: "Currently playing: Jake Bugg — Lightning Bolt". The "Currently playing" prefix is important for context.

Note that the politeness setting of the live region is `polite` not `assertive` as in the previous example. If the user is busy reading something else or typing, the notification will wait until they have stopped. It isn't important enough to interrupt the user, not least because it's their playlist: they might recognize all the songs anyway.

The `aria-atomic` attribute doesn't have to be used on the same element that defines the live region, as in Lemon's example. In fact, you could use `aria-atomic` on separate child elements within the same region. According to the specification:

> *When the content of a live region changes, user agents* **SHOULD** *examine the changed element and traverse the ancestors to find the first element with* `aria-atomic` *set, and apply the appropriate behavior.*
> — *Supported States and Properties*[124]

This means we could also include another block within our live region to tell users which track is coming up next.

```
<div aria-live="polite" role="status">

   <div aria-atomic="true">
     <h3>Currently playing:</h3>
     <p>Jake Bugg – Lightning Bolt</p>
   </div>

   <div aria-atomic="true">
     <h3>Next in queue:</h3>
     <p>Napalm Death – You Suffer</p>
```

124. http://www.w3.org/TR/wai-aria/states_and_properties#aria-atomic

```
    </div>

</div>
```

Now, when Jake Bugg's Lightning Bolt is nearing an end, we update the `<p>` within the next in queue block to warn users that Napalm Death are ready to take the mic: "Next in queue: Napalm Death – You Suffer". As Napalm Death begin to play, the currently playing block also updates with their credentials and at the next available juncture the user is reminded that the noise they are being subjected to is indeed Napalm Death.

ARIA-BUSY

I was a bit mischievous using Napalm Death's You Suffer as an example track because, at 1.316 seconds long, the world's shortest recorded song would have ended before the live region could finish telling you it had started! If every track was that short, the application would go haywire.

In cases where lots of complex changes to a live region must take place before the result would be understandable to the user, you can include the `aria-busy` attribute[125]. You simply set this to `true` while the region is busy updating and back to `false` when it's done. It's effectively the equivalent of a loading spinner used when loading assets in JavaScript applications.

Usually you set `aria-busy="true"` before the first element (or addition) in the live region is loaded or altered, and `false` when the last expected element has been dealt with. In the case of our music player example, we'd probably want to set a timeout of ten seconds or so, mak-

125. http://www.w3.org/TR/wai-aria/states_and_properties#aria-busy

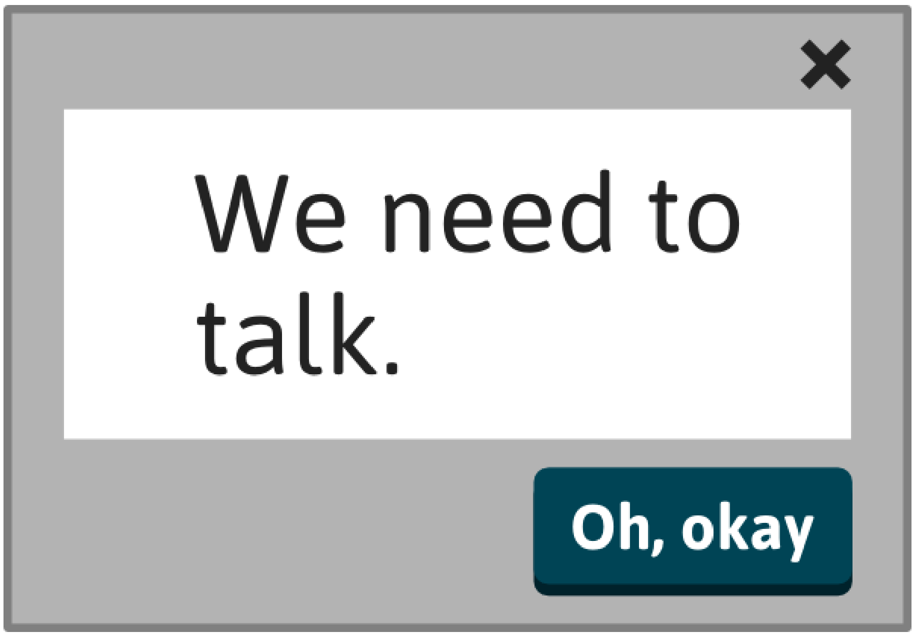
ing sure only music tracks longer than the announcement of those tracks get announced.

```
$('#music-info').attr('aria-busy', 'true');

// Update the song artist & title here, then...

setTimeout(function() {
   $('#music-info').attr('aria-busy', 'false');
}, 10000);
```

Dialogs

Accessibility engineers and evangelists the world over cringe at the suggestion of including dialogs in web application design. As with dreaded carousels[126], they are a UI enhancement that requires a great deal of consideration to make them usable for one user or the user sat next to them.

> **Dialogue**: *a conversation between two or more persons; also a similar exchange between a person and something else (as a computer)*
> — *Merriam-Webster definition*[127]



Unlike carousels, dialogs are an almost inescapably important pattern in desktop and web application design. As the name suggests, dialogs

126. http://www.creativebloq.com/accessibility-expert-warns-stop-using-carousels-7133778
127. http://www.merriam-webster.com/dictionary/dialogue

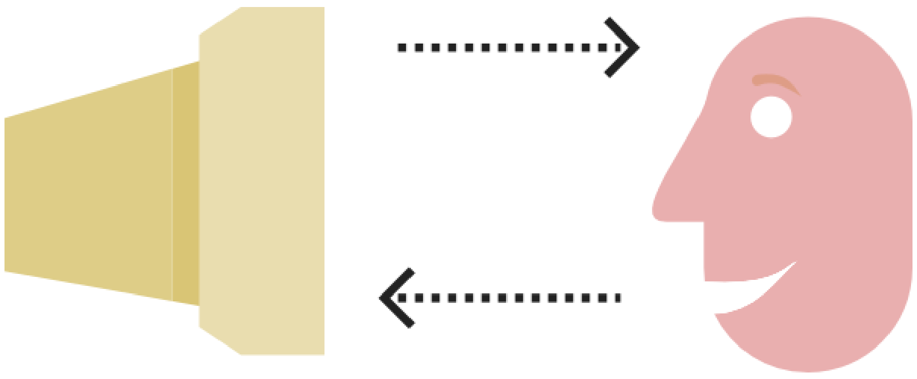
represent an impasse where an agreement between the user and the application must be reached before any other activity can take place. Dialogs usually take one of two forms:

- Some information that the user must read to keep them informed before they continue.
- A fork in the road; a choice of actions, one of which the user must take before doing anything else.

To keep things relatively simple for now, we are going to examine the cluster of techniques required to make a successfully accessible warning dialog. Despite paring things down, this will still require us to employ everything we've learned in previous chapters about semantic HTML, accessible roles, properties, and states *and* artfully managing and representing the focus of our controls.

THE ESSENTIALS

The dialog needn't exist until the impasse is reached, at which time we shall build it dynamically. Not only does this give us flexibility and remove redundant markup from the page, but conjuring it into existence also helps assistive technologies to recognize it as new, live content and treat it as a priority. The question is, what are the essential parts that make it a dialog?



A true dialog consists of just two parts: what one person (or computer, in this case) says; and what the other person says in response. To make our dialog pattern reusable, we are going to store these two variables in HTML5 `data` attributes (`data-dialog-call` and `data-dialog-response`) attached to any button control which might invoke a dialog. The benefit of `data` attributes[128] is that they are for pri-

128. http://ejohn.org/blog/html-5-data-attributes/

vate storage: like `<div>` elements, the meaning of `data` attributes is not revealed to the user. If it were, we would create unnecessary noise.

THE SETUP

Remember the **BIG RED BUTTON** from way back in chapter 2? Let's use that as the trigger for a warning dialog. After all, no good will come from pressing the **BIG RED BUTTON**, and the user — whoever they are — should probably be aware of that.

```
<button class="big-red" data-dialog-call="YI really don't like
you pressing that" data-dialog-response="I understand">Big Red
Button</button>
```

The <dialog> Element

As we shall soon discover, creating accessible web-based dialogs isn't the simplest of tasks. There's a lot to remember and a lot that can go wrong. This is why an easy-to-use programming interface for such dialogs is currently being specified.

It is already possible to enable standardized dialogs[129] using `<dialog>` and the element's associated attributes and methods in the experimental Chrome Canary browser and Chrome (versions 25+). By enabling experimental web platform features, you gain access to the following methods as part of the `HTMLDialogElement` interface:

- `show()`: shows a basic dialog.
- `showModal()`: shows an alert-like dialog and prevents the user from interacting with anything else on the page.
- `close()`: the method for properly closing a dialog.

With standardization comes clarity, meaning we no longer have to think twice about how a dialog should be made. However, it is not an interface which is ready for use in production yet and discussions[130] on how it will actually handle user behavior rage on.

For the time being, we can still use the `<dialog>` element so long as we register it as a known element via an HTML5 shim[131]. In addition, we can simulate the `open` attribute by attaching some CSS to it. Our

129. http://blog.teamtreehouse.com/a-preview-of-the-new-dialog-element
130. http://lists.w3.org/Archives/Public/public-html-bugzilla/2013Sep/0357.html
131. https://github.com/aFarkas/html5shiv/blob/master/src/html5shiv.js

script will toggle the `open` attribute directly to show and hide the dialog.

```
dialog {
    display: none;
}

dialog[open] {
    display: block;
}
```

Building the Dialog

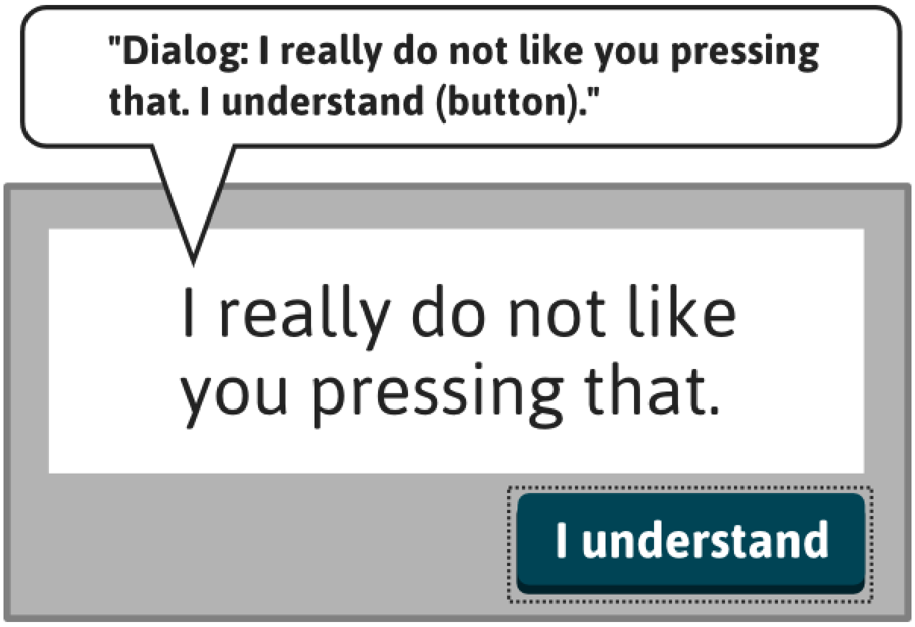
As mentioned, we shall be constructing the dialog dynamically, based on an inert `<dialog>` element. After running the script on the trigger's `click` function, the fully augmented `<dialog>` will look like this:

```
<dialog tabindex="0" open role="alertdialog"
aria-describedby="d-message">
   <div>
      <div>
         <p id="d-message">I really do not like you pressing
that</p>
         <button>I understand</button>
      </div>
   </div>
</dialog>
```

- **role="alertdialog"** overrides the native `<dialog>` semantics and defines the dialog as one intended for warnings and errors. Standard dialogs just take the `dialog` role. The `alertdialog` role is a special case live region[132] related to `alert`.
- **open** is just a Boolean attribute which says whether the dialog is open or — you guessed it — not.
- **tabindex="0"** is included so the `<dialog>` can be focused. On opening the dialog, we focus the close button, but making the dialog itself focusable will enable users to switch between the message and the close button to read either of them more than once if needed.

132. http://www.w3.org/TR/wai-aria-practices/#chobet

- **aria-describedby="d-message"** establishes a relationship between the dialog's message and the dialog, making sure the message is read whenever the `<dialog>` is focused.


"Dialog: I really do not like you pressing that. I understand (button)."
I really do not like you pressing that.
I understand

Using the free NVDA screen reader, opening the dialog reads "Dialog: I really do not like you pressing that. I understand (button)". That is, the dialog is announced for what it is, then the user is told what the dialog concerns, before finally being given their single option of compliance. Not all dialogs would offer just the one course of action, of course, but we're trying to keep things simple.

THE FOCUS CONUNDRUM

I've said it before and I'll say it again: managing the user's focus is one of the hardest and most important aspects of accessible application design. This is never truer than in the case of modal dialogs. There are three key areas we have to address:

1. Users must not be able to interact with elements outside the dialog while it is open.
2. Users' focus must not be trapped within the dialog entirely: they must be able to escape to the browser's address bar and other controls outside the page.
3. When closing the dialog, focus must be returned to the element (if any) which invoked it so they may continue where they left off.

Making all contents of the page except the dialog contents focusable is a bit of a head-scratcher all its own. First, we have to consider all the types of elements which may be focusable. Written as CSS selectors, these include:

- `a[href]`
- `button:not([disabled])`
- `[tabindex]:not([tabindex="-1"])`

as well as all types of form elements.

In the NC State University blog post "The Incredible Accessible Modal Dialog[133]",Greg Krauss describes a technique whereby all of these elements are identified, rendered not focusable, and stored in memory so that focus can be restored to each of them when the dialog is closed again. This feels inefficient, so perhaps it would be better to add a marker attribute on each, so only elements matching that marker are made focusable again.

```
<a href="http://www.google.com" tabindex="-1"
data-modal-unfocused></a>
```

To make matters more complex, we have to remember that these elements should also not be readable in a screen reader's browse mode. That is, unless they are hidden to screen readers, users could casually wander out of the dialog and continue reading whatever else is in the page.

The `aria-hidden` state will hide the contents of the page from most modern screen readers, so we ought to place `aria-hidden="true"` on all contents outside the dialog. Unfortunately, this will not stop keyboard users focusing elements, so we still have to control that separately. Seems like a lot of work. What if there's a better way?

The `visibility:hidden` Solution

You may recall discussion earlier in this book about how screen readers have a clever — though sometimes too clever — ability to apprehend certain CSS styles. We talked about how elements hidden with `display:none` or `visibility:hidden` are hidden both from view and from being read: equality.

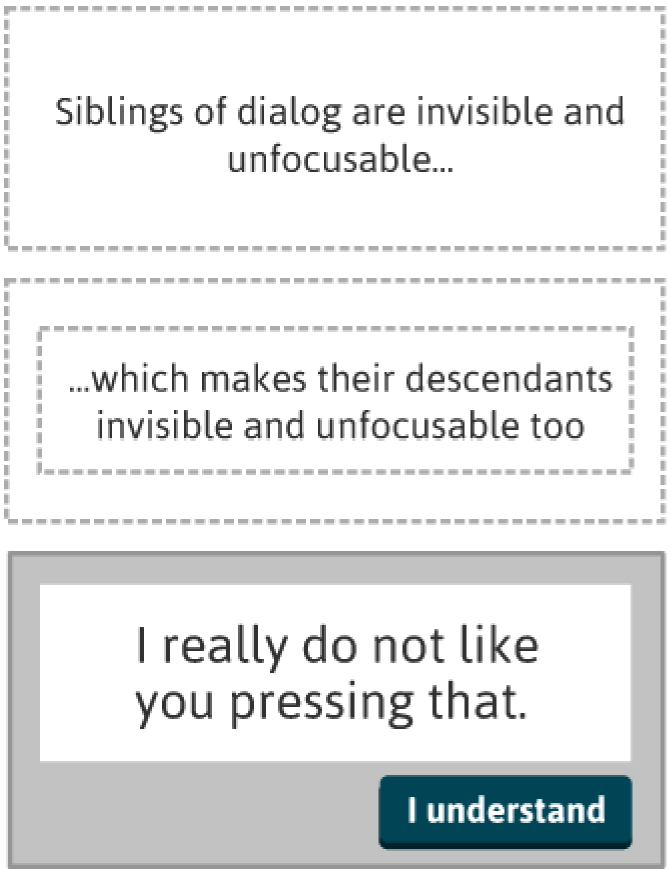
Well, elements hidden with `visibility:hidden` or `display:none` are also not focusable, making them unnavigable by keyboard. All we have to do is identify the contents of the page which aren't part of the dialog and attach a `class` attribute corresponding to the

133. http://accessibility.oit.ncsu.edu/blog/2013/09/13/the-incredible-accessible-modal-dialog/

visibility:hidden rule. (We choose visibility:hidden over display: none so that the elements retain their layout. We don't want to jog anything around.)

The following jQuery one-liner will add the mod-hidden class to all elements that are siblings of the <dialog> and will affect all of their descendent elements, too. For this to work, the <dialog> must be a direct child node of <body>, but that's a simple authoring task.

```
$('body > *:not(dialog)').addClass('mod-hidden');
```

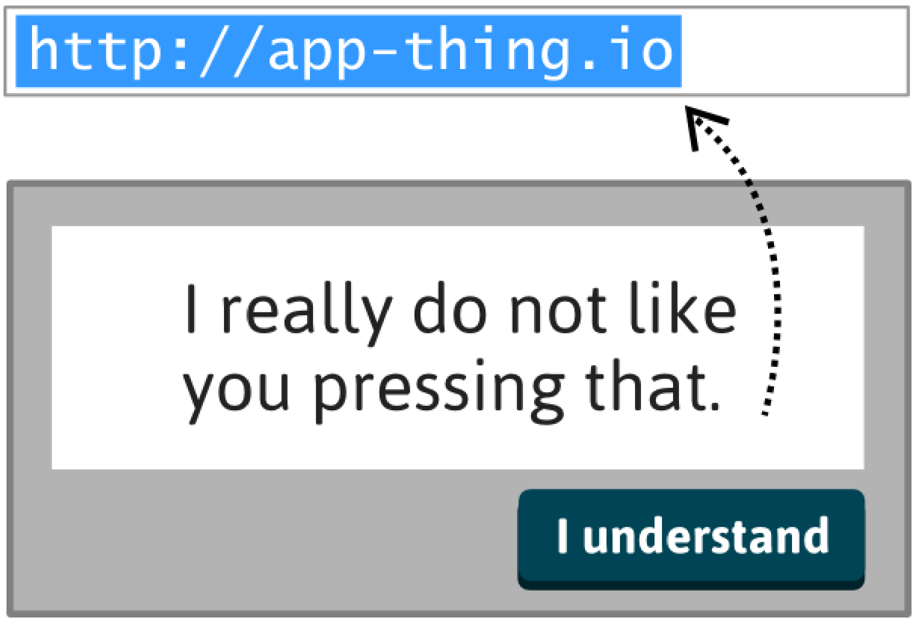

Siblings of dialog are invisible and unfocusable...
...which makes their descendants invisible and unfocusable too
I really do not like you pressing that.
I understand

It's a Trap!

To stop the user from escaping the dialog and interacting with other parts of the page before the dialog is attended to, we should make sure pressing *Tab* on the last focusable element within the dialog returns focus to the dialog itself. This can be done by overriding the close button's keydown handler:

```
close.on('keydown', function(e) {
    if ((e.keyCode || e.which) == 9) {
        dialog.focus();
        e.preventDefault();
    }
  });
```

In some implementations, pressing *Shift* + *Tab* on the dialog element would move focus back to the close button (or last focusable element in the dialog). However, creating this focus loop seals the user within the dialog, making it difficult for them to leave the page at all. In our version we shall let pressing *Shift* + *Tab* release the user to the browser's address bar, giving them an escape route.



THE CLOSE FUNCTION

The "I understand" button will run a function called, let's say, `closeDialog()` to close the dialog and remove all the generated markup. This should also be possible by pressing the *Esc* key; a feature which is advised for dialogs under "ARIA practices"[134].

```
$(dialog).on('keypress.escape', function(e) {
    if (e.keyCode == 27) {
      closeDialog();
    }
  });
```

Closing the dialog is as simple as removing the `open` attribute, but we must also remember to remove all the other attributes, leaving the `<dialog>` a simple placeholder for future dialog invocations. Importantly, we must return focus to the element which opened the dialog in the first place.

We can create an identifier for the trigger element using its `id` (or giving it a temporary `id` if it doesn't already have one):

134. http://www.w3.org/WAI/PF/aria-practices/#dialog_modal

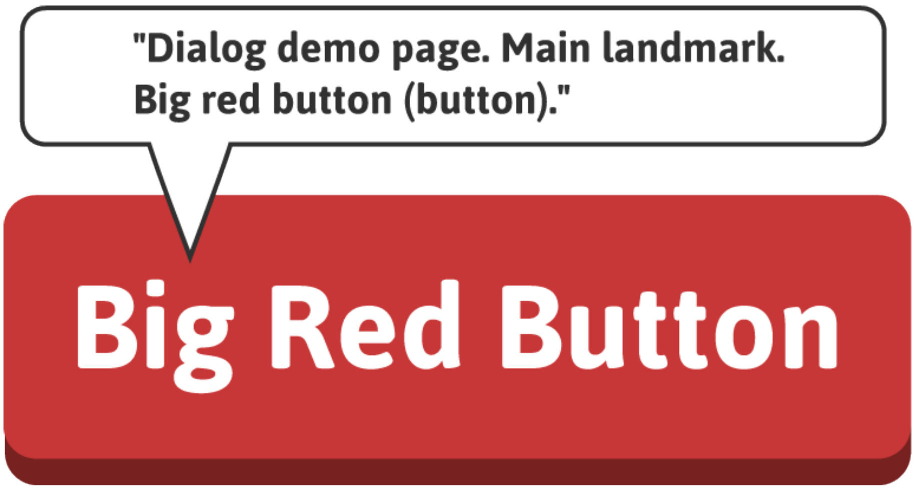
```
var trigger = $(this).attr('id') ? $(this).attr('id') : 'origin';
```

Then, as the last task in our closing process, we simply shift focus to it:

```
$('#' + trigger).focus();
```

The benefit of this is twofold, affecting both screen reader users and keyboard users. First, it reinstates the focus style on our **BIG RED BUTTON**, confirming it has focus and moving to it if it has slipped out of the viewport. Second, the reinstated focus is announced in context.

In the demo page[135] for this dialog example, the title of the page is announced, then the landmark location (main), followed by the button text. In a short sentence, the user has their location pinpointed.



You have been warned. Just don't press it again. Actually, do you know what? I have no idea why I put it there in the first place. It doesn't make any sense. ❧

135. http://heydonworks.com/practical_aria_examples/#warning-dialog

CHAPTER 7:

Welcome To The Community

I'm happy to report that learning about web accessibility has never been a lonely experience. The community of accessibility engineers, consultants and advocates I have been fortunate to correspond and meet with are particularly generous with their time and expertise — even when compared with the already open nature of the wider web community. Without conversing and collaborating with folk involved in web accessibility, this would be a much thinner — and frequently factually inaccurate! — book.

To conclude this introduction to web application accessibility, I'd like to introduce you to the accessibility community and many of the fantastic free tools and resources available to help you build accessible applications.

#a11y

In your travels around the web you may encounter the curious *numeronym* a11y. This is an abbreviation of the word "accessibility" where the "11" represents the number of omitted characters. In the accessibility community, the Twitter hashtag #a11y is frequently used as a sort of Bat-Signal to indicate that a resource has some bearing on accessibility practice. Sometimes #a11y is used as a distress signal, indicating a failure to address accessibility but, more often, it is used to draw attention to successes in the field of accessibility.

#a11y

I highly recommend you follow #a11y on Twitter to keep abreast of innovations in accessibility techniques, updates to specifications and improvements to the accessibility of popular applications and websites.

SOME FOLKS TO FOLLOW

Here are a few of the people I follow on Twitter for their expertise and experience regarding web accessibility. This is by no means a complete list of #a11y stalwarts. If you want to reach out further, check out who these folks follow too!

@stevefaulkner

Steve works for The Paciello Group, who are leaders in accessibility consultation. He is also an editor of the W3C's HTML specification, the innovator of the `<main>`, element and the technical reviewer of this book. Steve frequently writes for The Paciello Group's blog[136].

@LeonieWatson

Previously accessibility director at Nomensa, Léonie is now a colleague of Steve's at The Paciello Group and an influential accessibility advocate. A prolific user of the #a11y hashtag and owner of the blog, The Tink Tank[137].

@rogerjohansson

Roger Johansson has been involved in web development since 1994 and I have been reading his blog 456 Berea Street[138], which focuses on web standards and accessibility, since 2006.

@karlgroves

Karl is a tireless campaigner for accessibility. He blogs at karlgroves.com[139] and is looking for participants to help him build a CSS accessibility testing tool[140] much like the *test.css* we have been using here, but more comprehensive.

@jsutt

Jennifer Sutton is passionate about accessibility and has a deep knowledge of the subject. She helped to develop the W3C's guide to contacting organizations who have inaccessible websites[141].

---

136. http://blog.paciellogroup.com/
137. http://tink.co.uk/about-tink/
138. http://www.456bereastreet.com/
139. http://www.karlgroves.com
140. http://www.karlgroves.com/2013/09/07/diagnostic-css-super-quick-web-accessibility-testing/
141. http://www.w3.org/WAI/users/inaccessible

@dennisl

Dennis Lembrée is the founder and main host of the Web Axe[142] accessibility blog and podcast. You should follow @WebAxe too.

@jkiss

Jason Kiss is an advocate and researcher for accessibility whose detailed articles at Accessible Culture[143] leave no stone unturned in the search for demonstrably successful accessibility techniques.

@dboudreau

Denis Boudreau is another big name who, in addition to #a11y, also tweets with #a11yTips: a great tag to follow for easily digestible accessibility improvements.

@yatil

Eric Eggert is a seasoned accessibility specialist who works for the W3C, providing workshops and talks about web accessibility.

@aardrian

Adrian Roselli writes about usability mostly but our discussions about accessibility have always been interesting and fruitful. He is an invited expert at the W3C and writings in the accessibility category[144] of his blog are always lively.

@cookiecrook

I was happy to have a heated debate about dialog implementations with James Craig. He has helped author the WAI-ARIA specification and has a book available on progressive enhancement[145].

@MarcoInEnglish

Marco Zehe is Mozilla's accessibility quality assurance engineer. He also blogs at marcozehe.de[146] where his "easy ARIA tips" are excellent.

142. http://www.webaxe.org/
143. http://accessibleculture.org/articles/
144. http://blog.adrianroselli.com/search/label/accessibility
145. http://filamentgroup.com/dwpe/
146. http://www.marcozehe.de

@gezlemon

Gez knows *a lot* about accessibility. He is the owner of the blog, Juicy Studio[147], from which I pilfered the music player live region example for chapter 6.

@icaaq

Isac Lagerblad is a sighted screen reader user with a great deal of insight regarding HTML semantics.

@patrick_h_lauke

Patrick has been a great help to me as I've been grappling with some of the less transparent parts of the ARIA specification. He's currently doing a lot of research into touch events[148] and their accessibility implications.

@marcysutton

Marcy is a speaker and researcher who's been doing a lot of experimentation with accessibility regarding JavaScript frameworks like AngularJS and web components. Cutting-edge stuff.

@jared_w_smith

Jared writes for the WebAIM[149] resource I have alluded to frequently throughout this book. WebAIM has even curated its own list of accessibility people to follow on Twitter[150], so I'll let him take over from here.

Blogs

Some excellent blogs — many by the aforementioned Twitter users — to bolster your Feedly. Other RSS readers are available.

- The Paciello Group Blog[151]
- Humanising Technology Blog[152] (Nomensa)
- WebAIM[153] (Jared Smith)

147. http://juicystudio.com/articles.php
148. http://www.slideshare.net/redux/getting-touchy-an-introduction-to-touch-events-saint
149. http://webaim.org/blog/
150. http://webaim.org/blog/twitter-accessibility-roundup/
151. http://blog.paciellogroup.com/
152. http://www.nomensa.com/blog/

- WAI[154]
- Marco’s Accessibility Blog[155] (Marco Zehe)
- WebAxe[156] (Dennis Lembrée)
- Adobe Accessibility[157]
- Access Sites[158]
- Juicy Studio[159] (Gez Lemon)
- Accessible Culture[160] (Jason Kiss)
- Simply Accessible[161]
- Karl Grove’s Blog[162]
- The Tink Tank[163] (Léonie Watson)
- 456 Berea Street[164] (Roger Johansson)

Testing Tools

Again, not a complete list, but a selection of some of the best open source tools for testing the accessibility of web applications. It should be noted that using automated tools is necessary but *not* sufficient. An interface employing semantic HTML correctly is a good start, but this does not confirm that the application is usable. User Acceptance Testing[165] will help to give you a better indication.

153. http://webaim.org/blog/
154. http://www.w3.org/WAI/
155. http://www.marcozehe.de/
156. http://www.webaxe.org/
157. http://blogs.adobe.com/accessibility/
158. http://accessites.org/site/
159. http://juicystudio.com/
160. http://accessibleculture.org/
161. http://simplyaccessible.com/
162. http://www.karlgroves.com
163. http://tink.co.uk/
164. http://www.456bereastreet.com/
165. http://www.techopedia.com/definition/3887/user-acceptance-testing-uat

BROWSER EXTENSIONS

Accessibility Evaluation Toolbar[166]

This has long been a stand-by for me. Gives a good overview of potential accessibility problems in your HTML.

Author: Jon Gunderson

Dom Inspector (DOMi)[167]

This Firefox add-on is much like Firebug except it has the additional feature of indicating which parts of the DOM are actually accessible. By switching to "accessible tree" mode, only nodes (parts of the HTML) which are accessible are shown.

Juicy Studio Accessibility Toolbar[168]

Gez's tool has the neat feature of providing information about ARIA live regions.

Author: Gez Lemon

WAVE Toolbar[169]

Gives you a quick overview of issues and features (including ARIA features) using graphics added to the page design.

Author: WebAIM

Chrome Vox[170]

A fully fledged screen reader designed as an extension for Google's Chrome browser. Good support for ARIA and a worthy addition to any screen reader testing suite.

Author: Google

166. https://addons.mozilla.org/en-US/firefox/addon/accessibility-evaluation-toolb/
167. https://addons.mozilla.org/en-US/firefox/addon/dom-inspector-6622/
168. https://addons.mozilla.org/en-US/firefox/addon/juicy-studio-accessibility-too/
169. https://addons.mozilla.org/en-US/firefox/addon/wave-toolbar/?src=ss
170. https://chrome.google.com/webstore/detail/chromevox/kgejglhpjiefppelpmljglcjb-hoiplfn

WEB-BASED TOOLS AND BOOKMARKLETS

HTML Code Sniffer[171]

HTML Code Sniffer is a bookmarklet which reports errors, warnings and notices regarding WCAG 2.0 compliance. A nice interface and the ability to fork and extend.

Author: Squiz

accesslint.com[172]

A web-based accessibility evaluator. Just enter a URL.

Author: Cameron Cundiff

pa11y[173]

A browser-based dashboard for monitoring the accessibility of a number of your sites simultaneously. Requires node.js, MongoDB and PhantomJS.

FireEyes[174]

FireEyes is an extension for Firebug. It is a free and comprehensive reporting service which requires you to create an account but is free.

Author: Deque

WAVE[175]

The web-based alternative to WebAIM's browser extension.

Author: WebAIM

Diagnostic CSS[176]

A "super quick" accessibility testing tool using CSS only, much like our *test.css* file, but more complex and comprehensive.

Author: Karl Groves

171. http://www.squizlabs.com/general/html-codesniffer
172. http://www.accesslint.com/
173. http://pa11y.org/
174. http://www.deque.com/deque-fireeyes
175. http://wave.webaim.org/
176. http://www.karlgroves.com/2013/09/07/diagnostic-css-super-quick-web-accessibility-testing

REVENGE.CSS[177]

My own CSS-based testing tool. Warnings appear in pink boxes with text in Comic Sans. Inaccessible webpages *deserve* to look ugly (until they are fixed!)

Author: Heydon Pickering

CONTRAST AND COLOR BLINDNESS

colororacle.org[178]

A powerful desktop application for Windows, Mac and Linux which emulates typical color vision impairments. You can see, at a glance, whether certain parts of your design suffer from lack of definition.

Contrast Ratio[179]

A really easy-to-use contrast checker which allows you to enter a foreground and background color, and then returns a WCAG 2.0 compliance rating.

Author: Lea Verou

Contrast Checker[180]

Similar to Lea's tool above, but from WebAIM. Gives you the ability to increment and decrement the shade of either color (foreground or background) to see where the failure points are.

Author: WebAIM

Contrast Analyzer[181]

Another analyzer by the good people at The Paciello Group. Available for Windows and Mac OS X.

Colour Contrast[182]

A tool that takes the concept one step further, with sliders to tweak the colour components, including hue and saturation.

Author: Jonathan Snook

177. http://heydonworks.com/revenge_css_bookmarklet/
178. http://colororacle.org/
179. http://leaverou.github.io/contrast-ratio/
180. http://webaim.org/resources/contrastchecker/
181. http://www.paciellogroup.com/resources/contrastAnalyser
182. http://snook.ca/technical/colour_contrast/colour.html

WAI-ARIA

Some resources and resource round-ups specific to ARIA practice.

W3C SPECIFICATIONS AND GUIDELINES

- WAI-ARIA Overview[183]: as good a place as any to start
- WAI-ARIA 1.1[184]: the latest draft of the WAI-ARIA specification
- The Roles Model[185]: information on types of ARIA role and the relationships between different roles
- Supported States and Properties[186]: an introduction to the various state and property attributes that different roles support
- WAI-ARIA Authoring Practices[187]: includes step-by-step guidelines for creating ARIA widgets
- Using WAI-ARIA in HTML[188]: includes the "rules of ARIA use" which we covered early in the book. Note, this is only an *Editor's Draft*: a work in progress liable to change before it is published as an official *Working Draft*.

ARIA ON MOZILLA MDN[189]

A fantastic round-up of ARIA learning resources which I shall not try to compete with here. Just go and take a look.

PRACTICAL ARIA EXAMPLES[190]

All of the examples created for this book on one page, as well as one or two bonus widgets.

WAI-ARIA: INFORMATION AND EXAMPLES[191]

Another round-up, with some interesting examples not covered by this book, including calendars and drag-and-drop interfaces.

---

183. http://www.w3.org/WAI/intro/aria
184. http://www.w3.org/TR/wai-aria-1.1/
185. http://www.w3.org/TR/wai-aria-1.1/roles
186. http://www.w3.org/TR/wai-aria-1.1/states_and_properties
187. http://www.w3.org/TR/wai-aria-practices/
188. http://www.w3.org/TR/aria-in-html/
189. https://developer.mozilla.org/en-us/docs/web/accessibility/aria
190. http://heydonworks.com/practical_aria_examples/
191. http://wai-aria.punkchip.com/

And Finally...

It would be remiss of me not to mention the web application the Smashing Magazine team and I have been using to write and edit this very book. Editorially (which, regrettably will have been discontinued by the time this book has been published) has been an intuitive and enjoyable tool to use and — as it turns out — an impressively accessible one.

Having already drafted chapters of this book with Editorially, when a discussion about web-based content editors took place on Twitter, I was all too happy to recommend my new favorite toy. However, when Jennifer Sutton (an #a11y expert mentioned above and a blind screen reader user) asked me whether it was worth her bother, I wasn't sure what to say. Certainly, it was a simple application and simplicity is usually a good indicator of accessibility. But what if the simplicity I experienced was by virtue of the visual interface alone?

We exchanged emails and I told Jennifer I'd be interested in her opinion of the application. A few days later she replied with a glowing report. In fact, she found it so easy to use that — in her own words — "I almost fell out of my chair."

Of course, it wasn't perfect, but it was so good she wanted to meet with the creators to iron out the few remaining creases. No doubt in full knowledge that the app would soon be defunct, a co-creator agreed to meet with Jennifer regardless, for a face-to-face screen reader testing session. They wanted to learn more.

Over the course of this book we've examined some specific technologies and techniques that can help you make your web applications more accessible. I hope that you can refer to the book when you are unsure about the accessibility of particular design patterns and are hoping to iron out a few creases of your own. But remember the case of Editorially: the reason Editorially was accessible overall, despite some technical shortcomings, is because it offered a simple interface designed for people to get a task done painlessly. Some call this approach user experience design, others user-centered design. Since interfaces necessitate users, I think it's fair to just call it *good design*. In any case, designing for users, their preferences, and their limitations rather than for the whims of stakeholders, marketers, peers, or passing trends is the most direct route to accessibility. Take it. ❧

Credits

The Author

Heydon Pickering is a user experience designer, coder and writer (obviously) from Norwich in the United Kingdom. He is particularly interested in accessible design patterns and the relationship between form and function in web development. Heydon is a regular writer for Smashing Magazine, SitePoint and his own blog, heydonworks.com[192]. He's also a type designer and a couple of his free icon fonts are hosted on Font Squirrel[193].

The Technical Reviewer

Steve Faulkner is the senior web accessibility consultant and technical director, TPG Europe. He joined The Paciello Group[194] in 2006. He is the creator and lead developer of the Web Accessibility Toolbar accessibility testing tool. Steve is a member of several groups, including the W3C HTML Working Group and the W3C Protocols and Formats Working Group. He is an editor of several specifications at the W3C[195] including HTML 5.1[196], Using WAI-ARIA in HTML[197] and HTML5: Techniques for providing useful text alternatives[198]. He also develops and maintains www.HTML5accessibility.com[199]

The Cover

The web accessibility icon featured on the cover of this book was created by the author, Heydon Pickering, based on a design by The Accessible Icon Project[200]. The aim of The Accessible Icon Project is to provide a vigorous, dynamic alternative to the old International Symbol Of Access[201].

192. http://www.heydonworks.com
193. http://www.fontsquirrel.com/foundry/Heydon-Pickering
194. http://www.paciellogroup.com
195. http://w3.org
196. http://www.w3.org/html/wg/drafts/html/master/
197. https://dvcs.w3.org/hg/aria-unofficial/raw-file/tip/index.html
198. http://dev.w3.org/html5/alt-techniques/
199. http://www.HTML5accessibility.com
200. http://www.accessibleicon.org/about.html
201. http://en.wikipedia.org/wiki/International_Symbol_of_Access